The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

**Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.**

**To renew call Telephone Center, 333-8400**

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

MAR 17 1992

APR 28 1992

APR 03 1996

JAN 06 1997

L161—O-1096

Digitized by the Internet Archive
in 2013

http://archive.org/details/methodforconstru904sche

UIUCDCS-R-77-904

UILU-ENG 77 1756

# A Method for Constructing Compressed Parsers for a Parser Generating System

by

Richard Marion Schell, Jr.

November, 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

# A METHOD FOR CONSTRUCTING COMPRESSED PARSERS FOR A PARSER GENERATING SYSTEM

BY

RICHARD MARION SCHELL, JR.

A.B., University of Illinois, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Urbana, Illinois

510 84
I & Cr
no. 404-409
cop 2



ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor M.D. Mickunas for his guidance and patience as well as for his advice and helpful suggestions. I also owe my thanks to Alfred Weaver for his encouragement and for his part in getting me involved in this research.

Finally, my thanks go to my wife, Barbara, for her constant faith and unwavering patience.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION ..... 1

1.1. SURVEY OF METHODOLOGY ..... 3

1.2. OVERVIEW OF THE THESIS ..... 5

2. GRAMMARS AND ACCEPTORS ..... 6

2.1. DEFINITIONS ..... 6

2.2. NORMAL FORM GRAMMARS ..... 9

2.3. BOUNDED CONTEXT ACCEPTORS ..... 12

2.4. EXTENDED BOUNDED CONTEXT ACCEPTORS ..... 15

2.5. CONTEXT COMPUTATION AND CANONICAL COVERS ..... 18

3. TABLE REDUCTION STRATEGIES ..... 25

3.1. REPRESENTING W* ..... 26

3.2. MOTIVATION FOR IMPROVING THE TABLES ..... 27

3.3. TABLE TRANSFORMATIONS ..... 29

3.4. CONFLICT GRAPHS ..... 37

3.5. FINDING A BEST COMPRESSION SET ..... 43

3.5.1. ANALYSIS ..... 43

3.5.2. A BACKTRACK ALGORITHM ..... 47

3.6. TABLE ORDERING ..... 49

3.7. OBSERVATIONS ..... 54

4. THE PARSER GENERATING SYSTEM ..... 55

4.1. PGS INPUT ..... 55

4.2. THE STRUCTURE OF THE PGS ..... 56

4.3. TRANSFORMING THE INPUT GRAMMAR ..... 56

4.4. CONTEXT COMPUTATION ..... 61

4.4.1. ALGORITHMS ..... 61

4.4.2. REPRESENTING CONTEXT SETS ..... 64

4.5. CONFLICT GRAPH GENERATION ..... 65

4.6. STATE TABLE GENERATION ..... 66

4.7. ARRANGING ..... 72

4.8. ENCODING THE TABLES ..... 75

5. RESULTS AND CONCLUSIONS ..... 79

5.1. RESULTS ..... 79

5.2. CONCLUSIONS ..... 82

5.3. EXTENSIONS AND REFINEMENTS ..... 82

REFERENCES ..... 84

APPENDIX A ..... 87

APPENDIX B ..... 96

# CHAPTER 1

# INTRODUCTION

The subject of this thesis is a method for automatic parser generation. In particular, it explores a technique for producing compact and efficient parsers. The principal objective of generating parsers automatically is to take advantage of powerful parsing methods and at the same time to spare the compiler writer the burden of writing the parser. We will demonstrate techniques for generating parsers which satisfy this objective and which can be produced for a broad class of programming languages. And we will describe a parser generating system (PGS) that uses these techniques effectively.

The automatic generation of recognizers and parsers has been an area of continued interest since the late 1950's, when the prototypic "modern" programming languages were developed. Concurrent with the development of programming languages has been the development of theory and technology for recognizing and parsing these languages; this technology has resulted in the development of algorithms for constructing parsers mechanically. There have been several major issues faced in developing these methods. Among them are:

1. Allowing a broad class of languages to be specified easily. Some methods preclude desirable syntactic constructs from the set of languages they are capable of parsing; others require substantial effort on the part of the compiler writer to arrange his input grammar so that it is acceptable to the system.

2. Generating a useable parser. Parsers that are generated by mechanical systems must be both space and time efficient in order to meet the constraints of practical machine environments.

3. Generating a useful parser. Parsers must be integrated into the compiling process. To fit, they must provide capabilities for code generation and other semantic analysis and for detecting and repairing, or recovering, from programmer syntax errors.

Several methods have been developed for constructing parsers. To establish a perspective, we present some of the most representative methods.

## 1.1 SURVEY OF METHODOLOGY

The methods presented here are all deterministic bottom-up parsing methods. For a thorough survey, see [Aho 72] and [Feldman 68]. Mechanical generation of top-down parsers is rarely done, and top-down methods are inherently less powerful; consequently, LL parsing, though widely used, is not considered here.

Among the oldest methods for mechanical parser generation are the precedence parsing methods. [Floyd 63], [Wirth 66]. Such methods are conceptually simple and the generating algorithms are both simple and efficient. However, most precedence parsing techniques have undesirable characteristics. Not all programming languages are precedence parsable. Those that are often require substantial rewriting of the grammar for the grammar to be acceptable. Furthermore, the parsers produced are often large. The technique of Ichbiah and Morse [Ichbiah 70], which is based on weak precedence and uses Floyd-Evans productions [Floyd 61] and [Evans 64], is relatively space efficient but is still less general than other techniques available. Generally, precedence parsing is no longer a competitive method for generating parsers automatically. An extension to precedence parsing is mixed strategy precedence parsing; a parser generating system based on this method is in common use [McKeeman 71]. Again, the resulting parsers are large. And while it is theoretically possible to find grammars for languages in a broad class, in practice it is not always feasible to find one acceptable to both the system and to the compiler writer.

Currently, the most favored method in use is LR parsing [Knuth 65]. LR methods are powerful; the general class of LR parsers is capable of satisfying the requirements discussed previously. This is true also of the commonly used LR methods, LR(1), SLR(1) and LALR(1). Although LR parsing had obvious promise from the beginning, there were several disadvantages. Developing construction algorithms from the discussions in the early literature was difficult. When LR parsers were developed, the space required to store the parse tables was prohibitively large. Several independent researchers eliminated the latter problem, developing algorithms that produce parsers of a practical size. [Pager 70], [DeRemer 71] and [Lalonde 71]. Further research has demonstrated that LR construction algorithms can be made to produce very efficient parsers. [Joliat 73] and [Anderson 74]. Parser generating systems based on these techniques are widely available.

The method used by the PGS described here is a bounded context method. Bounded context parsing predates LR parsing [Floyd 63], [Eickel 63], but did not receive the same attention. Requirements on the input grammar tended to be very restrictive and the space requirements were also severe. A system for generating bounded context parsers was developed at Purdue University [Mickunas 74] which solved these problems; the system accepts reasonable input grammars and produces compact parse tables. The research and development reported here is a product of a modification to and extension of this system. The parsers produced are competitive with those produced by LR-based systems over a wide range of applications and is particularly

advantageous in cases where space is at a premium.

## 1.2 OVERVIEW OF THE THESIS

This thesis presents a description of the parser generating system. It also provides a detailed discussion of the techniques and algorithms used to minimize the size of the parsers it produces. Results and methods produced during the course of the thesis research are substantiated and proved. Remaining chapters are devoted to these tasks.

Chapter 2 provides background material. It introduces notation and terminology and introduces a formal model which is an abstraction of the parsing method used. Chapter 3 describes a process for constructing parse tables in a space-efficient manner. Methods and algorithms are introduced. Chapter 4 provides an overview of the PGS and shows how the methodology described in chapter 3 is integrated into the system. The chapter outlines the structure of the PGS and the implementation of the algorithms actually used. Chapter 5 presents results and summarizes the thesis presentation. It also suggests refinements and extensions.

There are two appendices; the first consists of four sections and illustrates the process of generating parse tables from an input grammar. The second is a full set of parse tables for an actual programming language.

# CHAPTER 2

# GRAMMARS AND ACCEPTORS

This chapter develops the mechanics of the parsing method used. In particular, it discusses bounded context grammars and a class of automata which accept languages defined by a subset of these grammars. It will be convenient to use terminology from formal language theory and parsing theory. We present some basic notation and terminology. The format follows Gray and Harrison [Gray 72], and Aho and Ullman [Aho 72].

## 2.1 DEFINITIONS

Definition. A (context-free) grammar (CFG) is a four-tuple,

$$G=(V,T,P,S)$$

where

V is a finite nonempty set of symbols, the vocabulary

$T \subseteq V$ is a finite set of symbols, the terminal vocabulary

$N = V-T$ is the nonterminal vocabulary

S is a distinguished element of N, the goal symbol

P is a finite subset of $N \times V^*$, the productions

We will denote an element (A,x) of P as A -> x. For any rule A -> x, A is called the left-part and x the right-part of the rule.

We will find it convenient to use binary relations on symbols.

Definition. Let $\rho$ be a binary relation on a set Y, $\rho$ Y X Y. Define $\rho^0$ to be the identity relation and $\rho^{i+1} = \rho^i\rho$ . Define the reflexive transitive closure $\rho^* = \bigcup\rho^i$; and the (non-reflexive) transitive closure $\rho = \rho^*\rho$. We will use $\rho u$ to denote $\{v \in Y \mid v\rho u\}$.

Definition. Let G = (V,T,P,S) be a CFG. For u, v in V* define u=>v if there exist x, w in V*, y in T* and A in N for which u=xAy, v=xwy and A -> w is in P. By this definition, we restrict derivations to rightmost, or canonical, derivations.

Definition. The set of (canonical) sentential forms for a CFG, G is denoted by CSF(G) and is defined as $\{x \in V^* \mid S=>^*x\}$. The sentences of the grammar are terminal strings in CSF(G). The language generated by G is the set of sentences of G.

Definition. If $x_i=>x_j$, then we say that $x_i$ directly derives $x_j$. If $x_0=>x_i=>\ldots=>x_r$, then we say that $x_0$ derives $x_r$; the sequence is called a (canonical) derivation of $x_r$ from $x_0$.

Definition. A CFG G is said to be unambiguous iff every sentence in the language of G has exactly one canonical derivation (from the goal symbol). A CFG which is not unambiguous is said to be ambiguous.

Definition. A string u in V* is a phrase if there is a derivation S=>*wAv=> wuv. If wAv=>wuv, then u is a simple phrase. The leftmost simple phrase of a sentential form is called the handle.

Definition. A CFG, G = (V,T,P,S) is said to be

1. e-free if $P \subseteq N \times V$ ,

2. reduced if for each A in V-{S} there are x, y in V* such that xAy is in CSF(G) and if no nonterminal is useless. A nonterminal is useless if it does not derive some terminal string.

For the remainder of the discussion, we insist that all grammars be reduced, e-free, and unambiguous. The following conventions for naming strings hold throughout the discussion:

A,B,C,D... are nonterminal symbols

a,b,c,d,... are terminal symbols

U,V,...,Z are either terminal or nonterminal symbols

u,v,...,z are elements of V*

## 2.2 NORMAL FORM GRAMMARS

The parsing method used is called bounded right context. Informally, a grammar is (m,n) bounded (right) context if given a sentential form in a (leftmost) parse, the handle can be determined by examining at most m symbols to the left and n to the right of a possible handle. The grammars used internally by the PGS are (1,1) bounded right context. It is proved in [Aho 72] that for every LR(k) language there is a (1,1)BRC grammar; further, given an LR(k) grammar, there is a mechanical transformation to a bounded context grammar of the specific form given by the definition below [Graham 74], [Mickunas 76].

Definition 2.1

A normal form grammar, G, is a 5-tuple

$G = (N,Q,T,S,P)$

where

N is a finite set of symbols called the pushdown or stack vocabulary.

Q is a finite, non-empty set of symbols disjoint from N called the state vocabulary.

T is a finite set of symbols called the input or terminal vocabulary.

$V = N \cup Q \cup T$ is called the vocabulary.

V-T is called the nonterminal vocabulary.

S is a distinguished element of Q called the goal or sentence symbol.

P is a finite subset of (N∪Q) X (N X Q ∪ Q X T ∪ Q ∪ T) called the production rules of G.

The production rules of G are of the following four forms:

1. p= E -> AB, called stack reducing rules,

2. p = E -> Ba, called input erasing rules,

3. p = E -> B, called renaming rules and,

4. p = E -> a, called initial state rules.

The grammars as used here are constrained so that right parts of all rules except renaming rules must be distinct; clearly, a grammar for which this condition does not hold can be transformed into one for which it does by coalescing productions and adding new nonterminals and renaming rules. This requirement can be relaxed but it is convenient to assume it holds in most cases.

Given a normal form grammar, a pair (A,a) in N X T is a context for a production p in P iff there is a derivation S =>+ xABay for which the last production is p. The stack symbol A in a context (A,a) is called the left context; the input symbol a is called the right context.

(1,1) bounded context grammars in this form have several important properties.

1. It is possible to determine the handle of a sentential form given a single context - the stack top and the next input symbol.

2. It is possible to distinguish between stack reducing (input erasing) rules on the basis of left (right) context alone.

3. Contexts can be contructed using a simple algorithm.

4. The parser and parse table structure are both simple.

The price paid for these properties is that the input grammar must be transformed into the specified internal form.

## 2.3 BOUNDED CONTEXT ACCEPTORS

The natural method of implementing a parser for a class of context free grammars is to contruct a pushdown automaton model which is realized as a table-driven parser. In this section, two separate automata are presented, both of which will parse using the grammars described in the previous section; the first is a simple model which corresponds naturally to the normal form grammars; the second is an extended model which properly includes the first and which serves as a good abstraction of the parsing method described in this thesis.

The first machine will be referred to as a bounded context acceptor, abbreviated BCA. As used in this thesis a BCA is a deterministic pushdown automaton represented by a sixtuple.

Definition 2.2

A bounded context acceptor is a 6-tuple

$$A = (N,Q,T,M,S_0,S)$$

where

N is the finite pushdown alphabet

Q is the finite, non-empty set of states

T is a finite set, the input alphabet,

M is a partial function from $N \times Q \times T$ into $(N \cup Q) \times Z$ where Z is the set of actions shift, reduce, and rename; M

is called the mapping of A.

$S_0$ is a distinguished element of Q called the initial state.

S is a distinguished element of Q called the accept state

A possible configuration for the automaton A is a triple (B,A,a) in N X Q X T; the configuration is valid if the mapping is defined for it. When M is defined it maps a configuration into a state-or-stack/action pair:

M(B,A,a) = (C,t),

where if t is a shift, the next input is consumed; if t is a reduce the topmost stack symbol is removed; and if t is a rename, no erasure occurs. If C is a state, the new state of the machine is C; if C is a stack symbol, then after the action t is performed, C is pushed onto the pushdown stack and the new state of the machine is the initial state.

An instantaneous description (ID) of the automaton A is a triple ($\sigma$,q,w) in N* X Q X T*; $\sigma$ represents the contents of the stack, q the state, and w the remaining (unscanned) input. The automaton A induces a relation $\vdash$ on the set of ID's as follows: ($\sigma$,q,w) $\vdash$ ( ',q',w') if

$\sigma$=uA and w=av and one of the following holds

1. $\sigma$'=u, w'=av and M(A,q,a)=(q',reduce)

2. $\sigma'=uA'$, $w'=av$ and $M(A,q,a)=(a',\text{reduce})$

3. $\sigma'=uA$, $w'=v$ and $M(A,q,a)=(q',\text{erase})$

4. $\sigma'=uAA'$, $w'=v$ and $M(A,q,a)=(a',\text{erase})$

5. $\sigma'=uA$, $w'=av$ and $M(A,q,a)=(q',\text{rename})$

6. $\sigma'=uAA'$, $w'=av$ and $M(A,q,a)=(a',\text{rename})$

The transitive closure $\vdash^*$ is well-defined. In order that the machine accept a string w, it is necessary that $(e,S_0,w) \vdash^* (e,S,e)$. (It is assumed that the machine is initially in state $S_0$ with an empty stack and ready to consume the first input symbol. If the input is well-formed, then the machine will stop in the accept state with an empty stack, having consumed all the input.)

The correspondence between the BCA and the normal form grammar for which the language that it accepts is obvious: given G=(N,Q,T,S,P), the corresponding BCA is

$$A=(N,Q \cup \{S_0\},T,M,S_0,S)$$

where M is constructed from P as follows:

1. If $A_i \rightarrow A_j a_k$ is in P, then for all of its left contexts $B_l$

define

$$M(A_j,B_l,a_k)=(A_i,\text{shift})$$

2. If $A_i \rightarrow B_jA_k$ is in P, then for all of its right contexts $a_l$ define

$$M(A_k,B_j,a_l)=(A_i,\text{reduce})$$

3. If $A_i \rightarrow A_j$ is in P, then for all of its contexts $(B_k,a_l)$ define

$$M(A_j,B_k,a_l)=(A_i,\text{rename})$$

4. If $A_i \rightarrow a_j$ is in P then for all of its left contexts $B_k$ define

$$M(S_0,B_k,a_j)=(A_i,\text{shift})$$

The correspondence between an instantaneous description of the BCA and a canonical sentential form is straightforward. For $A_1...A_r,A_ra_s...a_n$, if $A_r$ is in Q then the corresponding ID is $(A_1...A_r,,A_r,a_s...a_n)$. If A is in N, then the corresponding ID is $(A_1...A_r,S_0,a_s...a_n)$. That there is a one-to-one correspondence between the machine's actions and the steps in a derivation follows immediately.

## 2.4 EXTENDED BOUNDED CONTEXT ACCEPTORS

The acceptor defined in the previous section is sufficiently powerful to parse sentences from any deterministic context free lanugage. However, this does not completely serve; the second automaton

presented will allow for easier construction of good parse tables. This is accomplished by compressing the domain of the mapping function, which is a desirable end. Because of space considerations, the mapping function is implemented as a list and table lookup is used to determine the parse action to be taken. For the first automaton, if no space reduction is applied, the number of list items is the same as the total number of context pairs. By squeezing this domain, smaller initial tables will result.

The following definitions provide a compressed mapping:

Let W be a partial function from Q X powerset(N) X powerset(T) into powerset((N∪Q) X Z ∪ {w}). The symbol w in the range of W is used to indicate a special undefined value.

It will be said that W covers a BCA mapping M if:

whenever W is defined on a triple (A,U,V) then W(A,U,V) consists of the set of all M(A,B,a) for (B,a) in U X V; if there is some pair (B,a) in U X V for which M is not defined, then W(A,U,V) contains w.

and for all configurations (A,B,a) for which M is defined, the image M(A,B,a) is in some set in the range of W, W(A,U,V) for which B is in U and a is in V.

W is a deterministic cover of M if all elements in the range of W are singleton sets; W is a disjoint cover if when W(A,X,Y)= W(A,U,V) then either X and U are disjoint or Y and V are; W is a minimal cover if there is no proper restriction of W which covers the same mapping; finally, W is a minimum cover if there is no deterministic covering of the same mapping with a smaller domain. These definitions are sensible and realizable, in the sense that for any mapping a cover can be found which is characterized by one of the above definitions. Note that a mapping M can be embedded in the set of coverings of M in the natural manner; i.e. W(A,{B},{a})=M(A,B,a) whenever M(A,B,a) is defined and W is undefined otherwise.

An extended bounded context acceptor (EBCA) is a 6-tuple

$$\underline{A} = (N,Q,T,W,S_0,S)$$

where N, Q, T, $S_0$ and S are defined as with a BCA and W is a cover for a mapping M of a BCA $(N,Q,T,M,S_0,S)$. The notion of cover is extended to the EBCA by defining $\underline{A}$ to be a cover for A when N, Q, T, $S_0$, and S agree and W covers M. The actions of an EBCA are defined as follows:

In a configuration (A,B,a) the machine applies all possible actions which are in the sets W(A,U,V) for any U and V which contain B and a respectively.

Given an EBCA $\underline{A}$ which covers a BCA A, if the mapping for $\underline{A}$, W, is a deterministic cover for the mapping of A, then $\underline{A}$ is a deterministic pushdown automaton; furthermore, it accepts exactly the language of A.

That it is deterministic is easily seen: given any configuration (A,B,a), if there is more than one possible action, then there must be two set pairs whose cross product contains (B,a), say U,V and X,Y for which W(A,U,V) and W(A,X,Y) are not the same; this is not possible, since both image sets must contain M(A,B,a) and nothing else. That it accepts the same language is equally obvious: the actions applied for identical configurations are identical.

As a starting point for further space reduction, it would be most desirable to obtain an EBCA with minimum cover; this would produce the smallest number of entries in the unreduced table. Unfortunately, determining such a cover is an NP-complete problem and it is not desirable to use algorithms of such complexity. However, it is possible to generate suboptimal coverings with a simple polynomial algorithm, coverings which are still good in practice.

## 2.5 CONTEXT COMPUTATION AND CANONICAL COVERS

The cover used will be called the canonical cover; the sets for which the mapping is defined are computed during the context computation and no additional effort is required to extract them. In this section, the construction method is given. In order to simplify the discussion, the following binary relations are needed:

A $\lambda$ B if A -> Bu is in P

A $\rho$ B if A -> uB is in P

X $\alpha$ Y if A -> uXYv is in P.

The derived relations first, follow, and precede are defined by:

$$\text{first} = (\lambda^{-1})^*$$

$$\text{last} = \rho^*$$

$$\text{follow} = \text{first}\ \alpha^{-1}\ \text{last}$$

$$\text{precede} = \alpha\ \text{first}^{-1}$$

The set of initial elements of a binary relation $\beta$ corresponding to the second element x will be denoted $\beta(x)$. The sets so induced by the first, follow and precede relations will form the basis of the canonical covering.

In using the relations just defined, the following observations are useful:

1. X first A iff A =>* Xu . This follows immediately.

2. A precede B iff S =>* xABy .

   If A precede B, then there is a C such that C first B and a production p = D -> uACv. Since D is not useless, s=>*xABy. Conversely, if S=>*xABy then A precede B can be shown by induction on the number of productions in the derivation. Clearly, if S=>xABy in one step, then A precede B. Suppose S=>*xABy for a derivation of length k and for shorter derivations the hypothesis holds; obviously there is a final production in the derivation. There are three cases:

1. p = C -> w, B not in w. In this case, S=>*xABuCz in fewer steps, so by hypothesis, A precede B.

2. p = C -> uABv; this case follows immediately.

3. p = C ->Bu; in this case, S=>*xACw and A precede first B. By the definition of precede and the transitivity of first, it follows that A precede B.

3. a follow A iff S =>* xAay .

If a follow A then there are B and Y such that a first Y, B Y and B =>*uA. Since B Y, there is a production C-> wBYx in P and since a first Y, it follows that Y=>*av. Since the grammar is reduced, vx=>*z where z is in T*. Thus, C=>*wBaz and C=>*wuAaz for some string z. As C is not useless, the only if part follows.

Conversely, if S =>* xAay then a follow A, again by induction on the number of productions. If S=>xAay then a $\alpha^{-1}$ A. If S=>*xAay in k steps and for every shorter derivation the hypothesis holds, then there is a final production. There are four cases:

1. p = B -> vA; in this case s=>*wBay and so by hypothesis a follow B. Consequently, a follow ρ A and by the definition of follow and the transitiviy of last, a follow A.

2. p = B -> vAau. Here, $a\alpha^{-1}A$.

3. p = B -> au; in this case, S=>*xABy. So A precede B and B first A. It follows immediately that a follow A.

4. p = B -> w, where neither A nor a is in w. Since the derivation is rightmost, S=>*xAauBz in fewer productions.

There are three distinct classes of context sets these relations generate:

1. If p = E -> AB, then precede(E) X first(B) is a context set for every production used in any derivation A=>*C, C in Q.

2. If p = E -> Ba or p = E -> a, then precede(E) X {a} is a context set for the production p and for every production used in any derivation B=>*C, C in Q.

3. If p = E -> AB then {A} X follow(E) is a context set for production p and for every production in any derivation B=>*C, C in Q.

Contention: if (A,a) is a context for a production then it is contained in a set of one of the above types. This can be shown by cases:

case 1

Production p is of the type C -> Ba. If (A,a) is a context for p, then S=>*uACv=>uABav. Therefore A precede C and the set of class 2 covers this context. (Similarly for C -> a.)

case 2

Production p is of the type E -> AB. If (A,a) is a context for p then S=>*uEav=>uABv. Consequently a follow E. A class 3 set will cover (A,a).

case 3

Production p is of the type E -> B. If (A,a) is a context then there is an F such that S=>*uAFav=>*uAEav=>uABav. There are two subcases to consider. In the first, F is in Q. Clearly, (A,a) is a context for some production D -> Fa or D -> AF so it is covered by a class 2 or class 3 set. In the second, F is in N. In this case, there is a production in the derivation of the form G ->FD, so that S=>*uAGw=>uAFDw=>*uAEav=>uABav. This implies that A precede G and a first D, so (A,a) is covered by a class 1 set.

Using these facts, the covering for the canonical map W* is generated as follows:

1. if p = E -> Ba then W*(B,precede(E),{a})={(E,shift)}.

2. if p = E -> AB then W*(B,{A},follow(E))={(E,reduce)}.

3. if p = E -> B then:

   if E is in Q, then whenever W*(E,X,Y) is defined, define W*(B,X,Y) = {(E,rename)};

   and if E in N then W*(B,precede(D),first(F))={(E,rename)} for all D, F such that D -> EF.

4. For all other elements of Q X powerset(N) X powerset(T), W* is undefined.

The covering induced in this manner is suboptimal, but is still very good. The following observations justify that claim.

First, observe that there is exactly one context set for each stack reducing or input erasing rule. Second, observe that if E -> B is a renaming rule, then if there are two pairs of context sets U X V and X X Y such that

W*(B,U,V) = W*(B,X,Y) = {(E,rename)}

then if U X V and X X Y are not disjoint, there is a stack symbol F such that

F=>*E

$A_{11}$ -> F$A_{12}$

$A_{21}$ -> F$A_{22}$

and U = precede($A_{11}$), V = first($A_{12}$)

X = precede($A_{21}$), Y = first($A_{22}$).

This is not difficult to establish. Suppose either context set is class

2 or class 3. Then there are two distinct productions for some state symbol D, where D=>*E, for which U X V and X X Y are the respective context sets. But since they have non-empty intersection, there is a context (A,a) for which two separate productions can be applied. This violates (1,1) BRC.

# CHAPTER 3

# TABLE REDUCTION STRATEGIES

To make the extended BCA model derived in the previous chapter a useful tool, it is necessary to produce a table-driven parser and parse tables which allow the parser to simulate an EBCA for a given grammar. It is desirable to keep the space required by the parser as small as possible without sacrificing a great deal of speed efficiency. This chapter outlines techniques for producing reduced tables, parse tables produced from the canonical mapping by applying space reducing transformations.

As mentioned in the previous chapter, the acceptor mapping is implemented using a table lookup scheme. Such a scheme is obviously preferable to a matrix encoding of the original BCA mapping, which requires one entry for every possible configuration, including the invalid ones. It would require $|Q| \cdot |N| \cdot |T|$ entries to provide this representation. A table lookup method eliminates the need for the entries corresponding to erroneous configurations, at the expense of giving up the direct access of the matrix method. In the last chapter,

evidence was presented to show that the mapping W* is a good one. In this chapter, transformations are presented that produce efficient parse tables from the map W*. Algorithms which perform the transformations optimally will be given.

## 3.1 REPRESENTING W*

To implement W* in an economic manner, separate tables are maintained for each state of the EBCA; there are |Q|+1 separate tables to generate. These separate tables will be referred to as <u>state tables</u>. In this chapter, details of implementation will be ignored in general and the tables will be represented abstractly. An entry in the state table for a given state, B, coresponding to the mapping W*(B,U,V)= (E,t) will be written as $W^*_B(U,V)=(E,t)$; set brackets are eliminated. A state table is represented as an ordered sequence of such entries:

$$W^*_B(U_1,V_1)=(E_1,t_1)$$
$$\vdots$$
$$W^*_B(U_n,V_n)=(E_n,t_n).$$

Table lookup is performed as follows: given that the machine is in a configuration (A,B,a), the state table for state B is inspected one entry at a time starting with the first. There are two possibilities. If the configuration is valid, then there will be a smallest integer r such that $U_r$ X $V_r$ contains (A,a), in which case the action implied by $(E_r,t_r)$ is performed. On the other hand, if the configuration is

erroneous, all of the table entries will be exhausted without any action having been performed, in which case an error is detected.

The order of the entries in the table is irrelevant; it is only necessary that the entire state table be exhausted. Because of the deterministic nature of the EBCA, although there may be two entries for which a given configuration could cause an action to be applied, both actions applied would be identical. Later, this ordering property will be found to be advantageous.

## 3.2 MOTIVATION FOR IMPROVING THE TABLES

The representation as described in the previous section requires |dom W*| entries. If a fixed space requirement, or cost, is assumed for each entry, then the only improvement available is to find a better mapping, a possibility already rejected. However, there are good reasons to assume that the cost for each entry is not fixed. It is assumed that there are two possible sources of variable cost: the space required for the set pair; and the space required to encode the entry other than that required by the sets. It is further assumed that these are the only factors contributing to the cost of a state table other than the number of entries.

First, consider the set pair space requirement. It is reasonable to assume that each set will be encoded as a bit vector. Although these vectors might be of variable length, it is improbable that such an

encoding would provide much space economy, because of the overhead required to keep the vector lengths. Therefore, fixed vectors are assumed, thus requiring that the largest possible set must be accomodated. Hence, for each set of stack symbols, |N| bits are required and for each set of input symbols, |T| bits are required. If this space is maintained for each table entry, the space required for the sets alone would be prohibitively high. A more promising scheme is to maintain an auxiliary table of the distinct sets that are referenced and to use an index into this table in place of the actual set in the entries themselves. If this approach is adopted, the number of distinct sets is a source of variable cost, which should be made as small as possible. It can be assumed that singleton sets can be more economically encoded by using the index as the index of the symbol in the set of symbols.

The space required to encode an entry can vary because of the manner in which the inspection of an entry is performed. Given a configuration (A,B,a), the pair (A,a) is tested for membership in a set U X V by performing two tests: one for A in U and one for a in V. Thus, entries W* (U,V)=(E,t) for which U=N or V=T should require less space in the encoded table than other entries do; one (or more) of the memebership tests can be eliminated. For the rest of this chapter, let us stipulate that the cost overhead for each membership check is one unit; then the overhead required by an entry for a pair (U,V) is:

0 if U=N and V=T

1 if U=N or V=T, but not both,

2 if neither condition applies.

Acting on these assumptions, the cost of a set of parse tables can be reduced by minimizing the number of distinct sets referenced, by minimizing the number of expensive entries, and by reducing the number of entries. Such reductions will be done on a state-by-state basis because the task of global optimization is assumed to be intractible. The reductions are applied to tables generated from the canonical mapping to produce better tables; the latter may not correspond directly to any deterministic EBCA. The context sets for the reduced table entries may map into more than one acceptor action. Subsequently we will show this is allowable. The reduction schema are discussed in the remainder of this chapter.

## 3.3 TABLE TRANSFORMATIONS

The table reduction scheme consists of the application of two transformations to the W* tables. The first transformation is designed to reduce the number of expensive entries; the second is designed to merge entries, thereby diminishing the number of entries. It is assumed that by applying the first transformation, the number of distinct sets is also decreased. Consequently, no explicit transformations to decrease the collection of distinct sets are developed here. The transformed tables will be denoted as W*.

Transformation 1

Given a state table in arbitrary order,

$$W^*_B(U_1,V_1)=(E_1,t_1)$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$W^*_B(U_n,V_n)=(E_n,t_n).$$

the entry for $(U_i,V_i)$ can be transformed to $\overline{W}^*_B(U_i',V_i')=(E_i,t_i)$ where $U_i \subseteq U_i'$ and $V_i \subseteq V_i'$ whenever the action for any valid configuration $(A,B,a)$ where $(A,a)$ is in $(U_i'-U_i)$ X $(V_i'-V_i)$ will be applied for some entry r where $r<i$. In particular, the primed sets will be restricted so that $U_i'=N$ or $U_i'=U$ and $V_i'=T$ or $V_i'=V_i$. Further, the transformation is constrained by the following:

if $W^*_B(U,V)=(E,\text{reduce})$ then $U'=U$, and

if $W^*_B(U,V)=(E,\text{shift})$ then $V'=V$.

This transformation will be referred to as compression. An entry for which $U'=N$ will be called a left compressed entry and the corresponding transformation will be called left compression. An entry for which $V'=T$ will be called a right compressed entry and the corresponding transformation will be called right compression. An entry which is either left or right compressed is compressed. If it is neither left nor right compressed it is a full entry. If it is both right and left compressed it will be called an unconditional entry. Clearly, if there is an unconditional entry, it will be the last entry in the table, since

its actions will be performed for all configurations. The validity of this transformation will be shown after both transformations are given.

Transformation 2

Given $(U_r,V_r)$ through $(U_s,V_s)$ consecutively, the set pairs for state table entries $W^*_B(U_r,V_r)=(E_r,t_r)$ through $W^*_B(U_s,V_s)=(E_s,t_s)$ for which $(E_r,t_r)=(E_{r+1},t_{r+1})\ldots=(E_s,t_s)$ then if the $V_i$ are identical then the entire set of entries can be replaced by one entry

$$W^*_B(U_r \cup U_{r+1} \cup \ldots \cup U_s,V_s)=(E_s,t_s)$$

and likewise if all the $U_i$ are identical then the entries can be replaced by

$$W^*_B(U_s,V_r \cup V_{r+1} \cup \ldots \cup V_s)=(E_s,t_s).$$

This transformation is a simple merging process; a more general merging scheme would allow merging in certain cases in which neither all the $U_i$ nor all the $V_i$ are identical.

Transformation 2 will always be applied after transformation ; no further transformation is done thereafter. Both of these transformations are valid, in the sense that the acceptor implemented using the transformed tables will accept exactly the language accepted by the EBCA for which the mapping is represented by the initial tables. This obviously holds for the second transformation. The validity of the first transformation depends on the following two facts:

1. If $(A_{r-1},A_r,a_s)$ is a valid configuration, then given the ordered state table for state B, $W^*_B$, an action is performed on encountering the i-th entry $W^*_B(U_i,V_i)=(E_i,t_i)$. It is sufficient to show that if the table is transformed to $W^*$ by transformation 1, that the i-th entry will be inspected for the specified configuration. Obviously, if an action is applied for the t-th entry where $t<i$ then $(A_{r-1},a_s)$ is a member of $U'_t$ X $V'_t$; further, since the action was not applied for the set pair $U_t$ X $V_t$, $(A_{r-1},a_s)$ is in $(U'_t-U_t)$ X $(V'_t-V_t)$. But this violates the constraints of the hypothesis.

2. If $(A_{r-1},A_r,a_s)$ is not a valid configuration, then if some action is applied for the i-th entry, one of the following cases exists: the action erases the input, in which case $a_s$ is in $V_i$; the action reduces the stack, in which case $A_{r-1}$ is in $U_i$; or, no symbol is consumed. Consider the first case. Let $(A_1\ldots A_{r-1},A_r,a_s\ldots a_n)$ be the ID of the machine when the action is applied. In order to reduce the stack to empty, the machine must eventually enter some state in which $A_{r-1}$ is removed. Therefore, the following sequence of ID's is obtained.

$$(A_1\ldots A_{r-1},A_r,a_s\ldots a_n) \vdash^+ (A_1\ldots A_{r-2},A'_r{}'',a_s\ldots a_n)$$
$$\vdash (A_1\ldots A_{r-2},A'_r{}'',a_t\ldots a_n) \text{ or}$$
$$\vdash (A_1\ldots A_{r-2}A'_r{}'',S_0,a_t\ldots a_n).$$

In either case, if the stack is reduced $A'_r{}' \Rightarrow^* A_r u$, where $u=a_s\ldots a_{t-1}$ and $A'_r{}'' \rightarrow A_{r-1}A'_r{}'$. (From the correspondence between ID's and derivations.) Therefore, $A_{r-1}$precede $A_r$ so

that $(A_{r-1},a_s)$ is a valid context and the configuration $(A_{r-1},A_r,a_s)$ is also valid.

Consider the second case. To erase $a_s$, the following sequence of ID's must occur.

$$(A_1\ldots A_{r-1},A_r,a_s\ldots a_n) \vdash+ (A_1\ldots A_{q-1},A_q,a_s\ldots a_n)$$
$$\vdash (A_1\ldots A_{q-1},A'_q,a_{s+1}\ldots a_n) \text{ or}$$
$$\vdash (A_1\ldots A_{q-1}A'_q,S_0,a_{s+1}\ldots a_n).$$

where $A_1\ldots A_{q-1}$ is obtained from $A_1\ldots A_{r-1}$ by reducing the stack. There are two subcases to consider. Either $A_q=S_0$ or not. In the second case, it is clear that $A_q$ =>* $wA_r$. Futhermore, since the input is reduced in state $A_q$, there is a production $A'_q$ ->$A_qa_s$, so that $a_s$ follow $A_q$ and thus $a_s$ follow $A_r$. Consequently, $(A_{r-1},a_s)$ is a valid context for a production $A'_r$ -> $A_{r-1}A_r$. So $(A_{r-1},A_r,a_s)$ is a valid configuration. In the other case, consider the sequence of ID's which leads to a state in which $A_{q-1}$ is removed from the stack. If

$$(A_1\ldots A_{q-1},A'_q,a_{s+1}\ldots a_n) \vdash+ (A_1\ldots A_{q-1},A''_q,a_t\ldots a_n)$$
$$\vdash (A_1\ldots A_{q-2},A'''_q,a_t\ldots a_n) \text{ or}$$
$$\vdash (A_1\ldots A_{q-2}A'''_q,S_0,a_t\ldots a_n)$$

and there is no shorter sequence which reduces $A_{q-1}$, then $A'''_q$ -> $A_{q-1}A''_q$, and $A''_q$=>* $A'_qu$. It is clear that $A'_q$ -> $a_s$; therefore, $a_s$ first $A''_q$, implying that $a_s$ follow $A_{q-1}$. Hence, since $A_{q-1}$ =>* $wA_r$, $a_s$ follow $A_r$. As before, this leads to the conclusion that $(A_{r-1}A_r,a_s)$ is a valid configuration. In the

first two cases, we ensure that the stack is non-empty by adding an additional symbol to mark the bottom of the stack and an extra state which removes this symbol and accepts if and only if the input has been consumed.

In the third case, it is easy to see that eventually the machine reaches a configuration in which one of the first two cases applies. In any case, the parser may erase an arbitrary number of input symbols, and it may erase an arbitrary number of pushdown symbols, but eventually it will arrive in a state in which it can detect the error.

As an illustration of a transformation of the first type, consider a state B in which all the rules are input erasing rules:

$$C_1 \rightarrow Ba_1$$
$$\vdots$$
$$C_n \rightarrow Ba_n$$

with corresponding left context sets $U_1,\ldots,U_n$. The initial $W^*_B$ table is

$$W^*_B(U_1,\{a_1\})=(C_1,\text{shift})$$
$$\vdots$$
$$W^*_B(U_n,\{a_n\})=(C_n,\text{shift}).$$

For any i, if $(A_i,B,a_i)$ is a configuration, the only possible entry

which could be applied is the one for $W^*_B(U_i,\{a_i\})$. Applying transformation 1, the resulting table is

$$\overline{W}^*_B(N,\{a_1\})=(C_1,\text{shift})$$
$$\vdots$$
$$\overline{W}^*_B(N,\{a_n\})=(C_n,\text{shift})$$

Similarly, if there are only stack reducing rules in a state, then the corresponding situation exists. In states in which both types of rules exist, or in which renaming rules exist, applying transformation 1 is non-trivial. Consider the following simple case in which state B has one stack reducing rule and one input erasing rule:

$C_1$ -> $Ba_1$ with left context set $\{A_1,A_2\}$

and $C_2$ -> $A_3B$ with right context set $\{a_1,a_2\}$.

If the $W^*_B$ table is ordered as written, i.e.

$$W^*_B(\{A_1,A_2\},\{a_1\})=(C_1,\text{shift})$$
$$W^*_B(\{A_3\},\{a_1,a_2\})=(C_2,\text{reduce})$$

no transformations can be made. On the other hand, if the order is reversed, then the table can be transformed to

$$\overline{W}^*_B(\{A_3\},T)=(C_2,\text{reduce})$$
$$\overline{W}^*_B(N,\{a_1\})=(C_1,\text{shift}).$$

Clearly, the transformations which can be applied are dependent upon the order of the table entries. To simplify the criteria for applying transformation 1, the notion of <u>conflict</u> is introduced. Two entries

$W^*_B(U_i,V_i)=(E_i,t_i)$ and $W^*_B(U_j,V_j)=(E_j,t_j)$ for which $(E_i,t_i)\neq(E_j,t_j)$ are left conflicting if $U_i \cap U_j \neq \emptyset$; they are right conflicting if $V_i \cap V_j \neq \emptyset$. It is easy to see that two entries cannot be both left and right conflicting; otherwise the machine with mapping W* would not be deterministic.

Entries which correspond to shift or rename actions are candidates for left compression; it is possible to transform them to left compressed entries. Likewise, entries which correspond to reduce or rename actions are candidates for right compression. Given these definitions, the criteria for applying transformation 1 can be restated:

> An entry which is a candidate for left, repectively right, compression can be transformed into a left, respectively right, compressed entry exactly when the state table is ordered so that every right, respectively left, conflicting entry precedes the candidate in the ordering.

A set of entries for which there is an ordering of the table that allows all members of the set to be either left or right compressed is a compression set. Note that in general, the largest compression set is a proper subset of the set of state table entries. Consider the following set of table entries in state B:

$$W^*_B(\{A_1\},\{a_1\})=(C_1,\text{shift})$$
$$W^*_B(\{A_1\},\{a_2\})=(C_2,\text{shift})$$
$$W^*_B(\{A_2\},\{a_1\})=(C_3,\text{reduce})$$
$$W^*_B(\{A_2\},\{a_2\})=(C_4,\text{reduce})$$

It is easy to see that there is no ordering which satisfies the criteria given. In the following sections, we present schema for finding compression sets, table orderings and transformations; in the next section a convenient framework for developing these schema is presented, using tools borrowed from graph theory.

## 3.4 CONFLICT GRAPHS

To assist in understanding this section, we present some terminology and notation from graph theory. In general, the notation used is from Harary [Harary 69]. A graph, G=(X,E) consists of a finite nonempty set, X, of p vertices together with a (possibly empty) set, E, of q distinct unordered pairs of distinct vertices from X. Each pair in E is called an edge. An edge e={u,v} can be written as e=uv or as e=vu. If uv is an edge, then u and v are incident with the edge uv, and u and v are adjacent. A vertex v is an isolated vertex if it is not incident with any edges (adjacent to any vertices). A subgraph of a graph G is a graph G' all of whose vertices and edges are in G. If G' is a subgraph of G, we write $G' \subseteq G$. For a subset S of the vertices of G, the subgraph

induced by S, written as <S> or <S;G>, is the maximal subgraph (in the sense that no edge uv in G can be added to the subgraph) of G with vertex set S. A spanning subgraph of G, G', is a subgraph whose vertex set is the same as that of G.

The complement $\bar{G}$ of a graph G is the graph on the same vertex set in which uv is an edge if and only if it is not an edge of G. A complete graph is a graph in which every two distinct vertices are adjacent. The complete graph on p vertices is denoted $K_p$. An independent set, S, is a set of vertices no two of which are adjacent. Hence, if S is an independent set, <S;G> is a complete subgraph of G. The union of two graphs $G_1$ and $G_2$, denoted $G_1 \cup G_2$ is the graph whose vertex set is the union of the vertex sets of $G_1$ and $G_2$ and whose edge set is the union of the edge sets of $G_1$ and $G_2$.

A bipartite graph G is a graph whose vertex set X can be partitioned into two subsets $X_1$ and $X_2$ so that if uv is an edge of G, then u is in $X_1$ and v is in $X_2$ (or vice-versa). If every vertex in $X_1$ is adjacent to every vertex in $X_2$, then G is a complete bipartite graph. The complete bipartite graph with $|X_1| = m$ and $|X_2| = n$ is denoted $K_{m,n}$.

Obviously, a graph is a representation of a symmetric binary relation on a set of objects; here, the binary relation is the conflict relation developed in the previous section. To represent this relation, two graphs, called conflict graphs in this thesis, are defined.

Definition 3.1

A conflict graph pair for a set of state table entries is a pair of graphs

$G = (X, E_R)$, the right conflict graph,

$G = (X, E_L)$, the left conflict graph

such that X is in 1-1 correspondence with the set of table entries and for u, v in X

uv is in $E_L$ iff the entries corresponding to u and v are left conflicting

uv is in $E_R$ iff the entries corresponding to u and v are right conflicting

Whenever it is not ambiguous to do so, we will use the terms vertex and entry interchangeably. The relations $adj_L$ and $adj_R$ are defined such that u $adj_L$ v if uv is in $E_L$ and u $adj_R$ v if uv is in $E_R$.

As an example, the graph pair shown in figure 3.1 corresponds to the last set of table entries in the previous section. It is clear that $G_L$ is a spanning subgraph of $\overline{G}_R$ and vice versa, inasmuch as two entries cannot be both left and right conflicting.

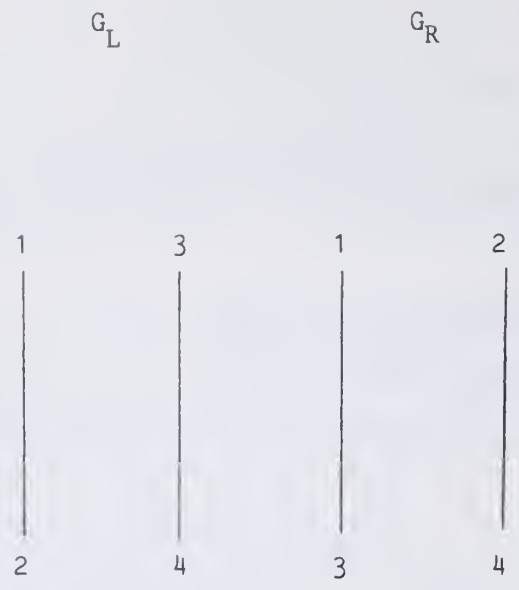



Figure 3.1

Graph pairs will be useful in visualizing conflicts, formulating algorithms and proving theorems. The first such theorem yields a simple test for detecting compressible sets given the conflict graph pair. The theorem requires the following simple definition: a <u>free vertex</u> in a subset S is a vertex which either corresponds to a left compression candidate and is an isolated vertex in $\langle S;G_R \rangle$ or corresponds to a right compression candidate and is an isolated vertex in $\langle S;G_L \rangle$.

Theorem 3.1

A subset Y of X, a vertex set for conflict graphs $G_L$ and $G_R$, is a compression set if and only if for every subset, S, of Y there is a free vertex in S.

Proof:

If. Since Y is a subset of itself, there is a free vertex, y, in Y. Let $Y_0 \supseteq Y_1 \supseteq \ldots \supseteq Y_n$ be a chain of subsets of Y such that $Y_0=Y$, $Y_1=Y-\{y\}$ and, in general, $Y_{i+1}$ is constructed from $Y_i$ by removing one of the free vertices in $Y_i$, say $y_i$. Corresponding to the sequence of $Y_i$, then, is a sequence of vertices, the $y_i$. Consider an arbitrary $y_i$. Without loss of generality, we can assume it to be a left compression candidate and isolated on $\langle Y_i;G_R\rangle$. Since it is isolated on $\langle Y_i;G_R\rangle$, there is no $y_j$, $j>i$, which is right conflicting-- otherwise $y_iy_j$ would be an edge of $\langle Y_i;G_R\rangle$. From this, it is easy to see that the sequence $y_0,y_1,\ldots,y_n$ corresponds to an ordering of the state table which allows simultaneous compression of the entries in Y.

Only if. Let U be a subset of Y containing no free vertex. For any ordering of the vertices of Y, say $y_1, y_2, \ldots, y_n$ there is a smallest j for which $y_j$ is in U. Suppose $y_j$ corresponds to a left compression candidate; then there is $y_j y_k$ in $\langle U; G_R \rangle$. By the assumption that j is smallest, k>j. Thus there is a conflicting vertex not preceding $y_j$. A similar result holds if $y_j$ corresponds to a right compression candidate. Since the ordering was arbitrary, it follows that there is no ordering for which every entry left conflicting with a left compression candidate precedes it and every entry right conflicting with a right compression candidate precedes it.

A set which does not contain a free vertex will be called an irreducible set. The theorem just given provides the following algorithm, which tests a set Y for being a compression set.

Algorithm 3.1

Input: A pair of conflict graphs, $G = (X, E_L)$ and $G = (X, E_R)$ together with the set of left compression candidates, L and the set of right compression candidates, R.

Output: An irreducible subset of X.

Method: L and R are searched until a free vertex is found; the vertex is removed from the sets and the algorithm repeats. If no free vertex is found, the algorithm halts, returning L R.

(1) Set Y=X.

repeat

(1) find a vertex, y, in $L \cap Y$ such that $adj_R y \cap Y = \emptyset$

(2) if (1) fails, find a vertex in $R \cap Y$ such that $adj_L y \cap Y = \emptyset$

(3) if either (1) or (2) succeeds, remove y from Y;
else return Y and halt

The algorithm presented can be improved upon; a better algorithm is presented in the next chapter.

## 3.5 FINDING A BEST COMPRESSION SET

### 3.5.1 ANALYSIS

Although it is easy to test for the property of compressibility, it is not easy to construct an efficient method for constructing a largest possble compression set, as will be shown. First, it is necessary to define what we mean by efficient. It is customary to say that an algorithm is efficient only when the time required to solve a problem is

bounded by a polynomial function of the size of the problem. Therefore, a method which examines every subset of the input set and tests for compressibility is inefficient; it requires time exponential in the size of the input set. Unfortunately, the problem of finding the largest compression set belongs to a class of problems called non-deterministic polynomial-time-complete (NP-complete) problems. An NP-complete problem is one for which there is a nondeterministic polynomial-time algorithm and which is as hard as any other problem for which there is such an algorithm. That is, if there is a deterministic polynomial time algorithm for the problem, then there is one for every NP problem. For a discussion of NP-complete problems, see [Karp 72], [Aho 74]. In order to show that finding a largest compression set is NP-complete, it is necessary to show that there are conflict graphs corresponding to any arbitrary pair of graphs $G_1$ and $G_2$ such that $G_1 \subseteq \overline{G}_2$ and $G_2 \subseteq \overline{G}_1$.

Lemma

If $G_1$ and $G_2$ are arbitrary graphs such that $G_1 \subseteq \overline{G}_2$ and $G_2 \subseteq \overline{G}_1$, then there is a corresponding pair of conflict graphs $G_L$ and $G_R$ such that $G_1 = G_L$ and $G_2 = G_R$. (The roles of $G_L$ and $G_R$ are reversible.)

Proof

It is easily seen that $G_1$ and $G_2$ must have the same vertex set X. Let the vertex set, X, of $G_1$ and $G_2$ be $x_1, x_2, \ldots, x_n$. Construct a set of table entries

$$W^*_B(U_1, V_1) = (E_1, \text{rename})$$
$$\vdots$$
$$W^*_B(U_n, V_n) = (E_n, \text{rename})$$

such that if $x_i x_j$ is an edge of $G_1$, $U_i \cap U_j \neq \emptyset$ and if $x_i x_j$ is an edge of $G_2$, then $V_i \cap V_j \neq \emptyset$. This situation is trivially constructible: let $A_{11}, \ldots, A_{nn}$ be elements of N and $a_{11}, \ldots, a_{nn}$ be elements of T. $U_i$ and $U_j$ contain $A_{ij}$ exactly when $x_i x_j$ is an edge of $G_1$ and $i<j$. Similarly, $V_i$ and $V_j$ contain $a_{ij}$ exactly when $x_i x_j$ is an edge of $G_2$ and $i<j$. Since $x_i x_j$ in $G_1$ implies that $U_i \cap U_j \neq \emptyset$, $G_1 \subseteq G_L$. (Similarly, $G_2 \subseteq G_R$.) On the other hand, if $U_i \cap U_j \neq \emptyset$, then it must be that $A_{ij}$ is in $U_i \cap U_j$; thus, $G_L \subseteq G_1$ (and $G_R \subseteq G_2$).

Theorem 3.2

The problem of finding a largest compression set is NP-complete.

Proof

First, we wil show that the problem is NP-hard by showing that a known NP-complete problem can be transformed to it in polynomial time while making the size of the input a polynomial function of the size of the original input. It is known that the problem of finding the largest complete subgraph of a graph is NP-complete [Aho 76]. It is, therefore, equivalently hard to find the largest independent set of a graph. We show by construction that if there is an algorithm to find the largest compression set of a state table, then it can be used to find the largest independent set of a graph.

Given a graph G on a vertex set of size p, construct the graph pair

$G = G \cup K_p$, where the vertex set X' of $K_p$ is disjoint from X

$G = K_{p,p}$ where the two vertex sets are X' and X.

Suppose the largest independent set of G is of size k. Assuming all vertices represent candidates for both left and right compression (renaming actions), the size of the largest compression set is exactly p+k. Certainly, there is a set which is this large. Take a largest independent set of G, S. Then by theorem 3.1 $S \cap X'$ is a compression set, since every vertex of S is free on any subset of $S \cap X'$ containing it, and every vertex of X' is free on any subset of X'. On the other hand, if Y is a compression set, it

obviously cannot contain more than p vertices from X'. Further, it cannot contain more than k vertices from X. For suppose it did. Then $X \cap Y$ must contain at least k+1 vertices. Consequently there must be two vertices in $X \cap Y$, say u and v such that uv is in G. In that case, $\{u,v\} \cup X'$ is an irreducible set.

That the problem is NP is easy to see: an enumeration algorithm which contructs and tests each subset of the vertex set requires polynomial time if it is run on a nondeterministic machine.

It should be noted that it is an unsettled question whether there are polynomial-time algorithms for NP-complete problems. It is conjectured that this is not the case, based on the fact that no such algorithms have been found for a number of independent problems. Therefore, while it is marginally possible that there is an efficient algorithm to find a largest compression set, it does not seem likely.

### 3.5.2 A BACKTRACK ALGORITHM

The following algorithm determines the largest compression set for a given state table by partial enumeration. The foundation for this algorithm is the simple fact that if U and U' are subsets of a vertex set X of a conflict graph pair such that: $U' \subseteq U$; U induces an irreducible configuration; and U' is a compressible set, then there is a vertex in U' which is not isolated in one of $\langle U;G_L \rangle$ or $\langle U;G_R \rangle$ but is isolated in the corresponding subgraph induced by U'. For example,

consider vertex 3 in figure 3.2.

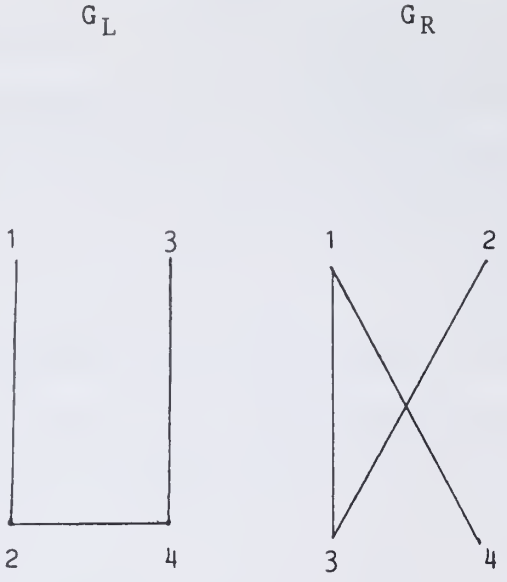



Figure 3.2

Algorithm 3.2

Input: A pair of conflict graphs, $G = (X, E_L)$ and $G = (X, E_R)$ together with the set of left compression candidates, L and the set of right compression candidates, R.

Output: The largest compression set for the graph pair.

Method: C is a function which returns the largest compression set. It accepts as input parameters U and D, where U is a subset of vertices which contains the remainder of the

compression set; and D is a set of vertices which can be discarded from U when an irreducible configuration is discovered. The result returned by C must contain D or be the empty set.

C(U,D) is given by:

(1) Find an irreducible subset of U, V (see algorithm 3.1).

(2) if V=ø, return U

(3) let x be a vertex in the smallest set contained in

{S | S⊆D & ((S=adj$_L$ v∩R)(v∈V∩L) or (S=adj$_R$ v∩L)(v∈V∩R)) }

(if the set is empty, return ø)

(4) recur, letting A=C(V-{x},D-{x})

(5) if A≠V then recur, letting B=C(V,D-{x})

else return A∪(U-V)

(6) return the larger of A∪(U-V) and B∪(U-V)

## 3.6 TABLE ORDERING

Finding a largest compression set only solves part of the problem; it is desirable to find a good ordering of the table entries. It is important especially to collapse the final, unconditional entries in the table. It is also important to order the tables to take advantage of the merging transformation. There are several possible strategies for

ordering the compressible set to obtain good merging to be considered.

The first strategy is a simpleminded one. After finding a compressible set Y, algorithm 3.1 is applied to the set. The order in which the algorithm removes free vertices is used to generate the table ordering. The algorithm is clearly suboptimal; the arbitrary selection of the next vertex to remove can eliminate possible unconditional entries or prevent other merging. For example, consider the following state table:

$$W^*_B(\{A_1\},\{a_1\})=(E_1,\text{rename})$$

$$W^*_B(\{A_2\},\{a_2\})=(E_1,\text{rename})$$

$$W^*_B(\{A_3\},\{a_2\})=(E_2,\text{reduce})$$

$$W^*_B(\{A_1\},\{a_3\})=(E_3,\text{shift})$$

The conflict graphs corresponding to this table are shown in figure 3.3.

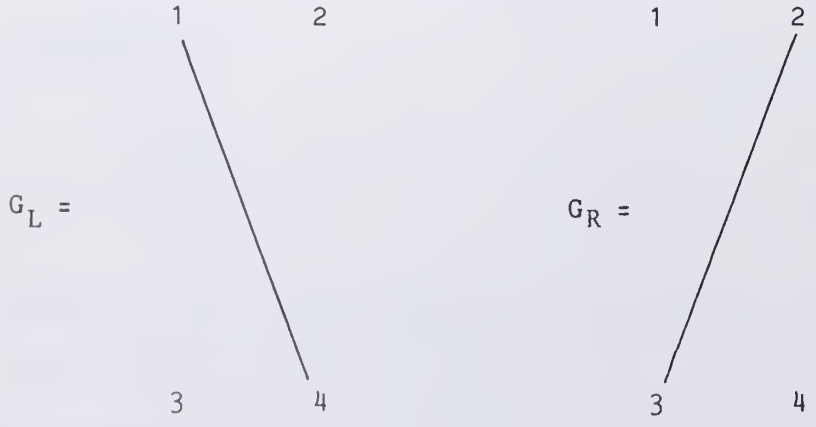



Figure 3.3

Notice that all four vertices are free. The ordering (1,2,3,4) is equally as likely as any other. This ordering results in the following transformed table.

$$\bar{W}^*_B(N,\{a_1\})=(E_1,\text{rename})$$
$$\bar{W}^*_B(\{A_2\},T)=(E_1,\text{rename})$$
$$\bar{W}^*_B(\{A_3\},T)=(E_2,\text{reduce})$$
$$\bar{W}^*_B(N,\{a_3\})=(E_3,\text{shift})$$

If an optimal strategy is used, the table can be transformed to

$$\bar{W}^*_B(\{A_3\},T)=(E_2,\text{reduce})$$
$$\bar{W}^*_B(N,\{a_3\})=(E_3,\text{shift})$$
$$\bar{W}^*_B(N,T)=(E_1,\text{rename})$$

Obviously, the strategy can be improved by employing heuristics in determining which vertex is to be removed. A polynomial-time suboptimal algorithm will result, but one which can avoid the obvious bad orderings such as the one just given. Such a method is actually used in the implementation.

If it is sufficient to find an ordering which exploits only the collapsing of unconditional entries, then there is an optimal method for doing this which requires polynomial time. As illustrated in the last

example, the renaming rule entries all corresponded to free vertices in the entire vertex set. The decision to choose vertices 3 and 4 allowed the two renaming rule entries to be output last and consequently to be collapsed into one unconditional entry. It is clearly desirable to collapse a large number of entries in general. The following lemma shows that there is a simple method for determining the optimal ordering to do this, assuming that the renaming rule to which the unconditional entry corresponds has been determined.

Lemma

Let Y be a compression set. Suppose there is an ordering of entries such that a set of vertices, S, corresponds to a group of unconditional entries. Then if U is a subset of Y containing an element not in S, there is a free vertex in U that is not in S.

Proof

Suppose not. Then there is a subset U such that some x in S is the only free vertex. There is some ordering of the table entries corresponding to vertices in Y that allows a compression transformation. Let $S_L$ be the set of left compressed entries and $S_R$ the set of right compressed entries in the resulting transformation. Now, since discarding entries cannot create

additional conflicts, it is clear that there is an ordering of the entries in U such that vertices in $S_L$ U are left compressible and those in $S_R$ U are right compressible. Let that ordering on U be $y_1, y_2, \ldots, y_n$. Then $y_1=x$, because x is the only free vertex in U. Also, $m>1$, since $|U|>1$ by the hypothesis. Let $y=y_2$. Without loss of generality, assume $adj_L x$ $U=\emptyset$. Then y $adj_R$ x, otherwise it is free on U. Further, y is not isolated on $\langle U;G_L\rangle$; if it were, then removing y from U-{x} would not create a free vertex. Therefore, y is in $S_L$. But since x is in S, all right and left conflicting entries precede it so that $S_L$ $adj_R x=\emptyset$. This leads to an obvious contradiction.

This lemma implies that if, whenever there is a choice of free vertices to remove, the algorithm always tries to remove one not in a given set of vertices corresponding to the same (renaming) rule, then the largest possible subset of that set will be left to be transformed to the unconditional entry.

Algorithm 3.3

Input: Y, a compressible subset of a vertex set X, with subsets L and R, the left and right compressible entries.

Output: Ordered sequences of elements of Y, representing table orderings such that all entries can be compressed.

repeat for all pairs (E,rename) for entries in the state

(1) let Y be the compression set

(2) repeat

(a) find all vertices, y, in $L \cap Y$ such that $adj_R y \cap Y = \emptyset$

(b) find all vertices, y, in $R \cap Y$ such that $adj_L y \cap Y = \emptyset$

(c) among all the vertices found in both (a) and (b), if there is one corresponding to an entry with action not (E,rename) output that vertex and remove it from Y.

(d) if (c) fails, output any vertex found and remove it from Y.

Algorithm 3.3 will not, in general, produce an ordered state table which can be transformed into an optimal state table. It is doubtful that there is a polynomial-time algorithm to generate an ordering which allows optimal application of transformation 2.

## 3.7 OBSERVATIONS

As a general observation, the algorithms presented here are too expensive to apply; they utilize too much computing time. In the implementation, ad hoc approximations to them were used, with reasonable success. Comparison of results obtainable using the exact methods should be done, in order to determine the worth of using them in restricted cases. Such comparisons are not covered by this thesis.

# CHAPTER 4

# THE PARSER GENERATING SYSTEM

This chapter represents a departure from the formal presentation of the previous two; it outlines an implementation specification for a parser generating system that uses the methods presented in the previous chapters. The structure of the system, the stages of generation and the interfaces between the stages are detailed. Alternate and supplementary algorithms are given to complete the development started in chapter 3.

## 4.1 PGS INPUT

Input to the PGS is an augmented BNF; the form of the input and restrictions to it are discussed in section 4.3. The PGS outputs parse tables which are essentially an encoded, compressed representation of the EBCA mapping described earlier. However, because the tables are used to drive a parser embedded in a compiler, and not just to drive a recognizer, it is necessary to provide information to the rest of the compiler. Semantic actions are included in (some of) the table entries; these are used to generate code or to label abstract parse trees (or to perform similar semantic functions.)

## 4.2 THE STRUCTURE OF THE PGS

There are six phases of the PGS which transform user input into parse table output. The first transforms the user input grammar, an augmented form of BNF, into an internal grammar in the normal form described in chapter 2. The second phase uses the internal grammar to compute context sets and attaches the context information to the productions of the internal grammar. Phase three generates conflict graphs for each state from the context information. Phase four uses the conflict graphs to generate properly ordered, compressed state tables. Phases three and four are applied successively on a state-by-state basis. Phase five arranges the tables to gain additional space savings and arranges the initial state table to make it possible to do direct lookup. (See section 4.5.) The final phase encodes the state tables in a usable form, which may vary to allow the generation of parse tables for different host machines. The diagram, figure 4.1, shows the flow of data through the six phases.

## 4.3 TRANSFORMING THE INPUT GRAMMAR

The first phase accepts augmented BNF and transforms it to normal form. The transformations were developed in [Mickunas 73]. For completeness, they will be described, albeit briefly. For more detail, see [Mickunas 76].

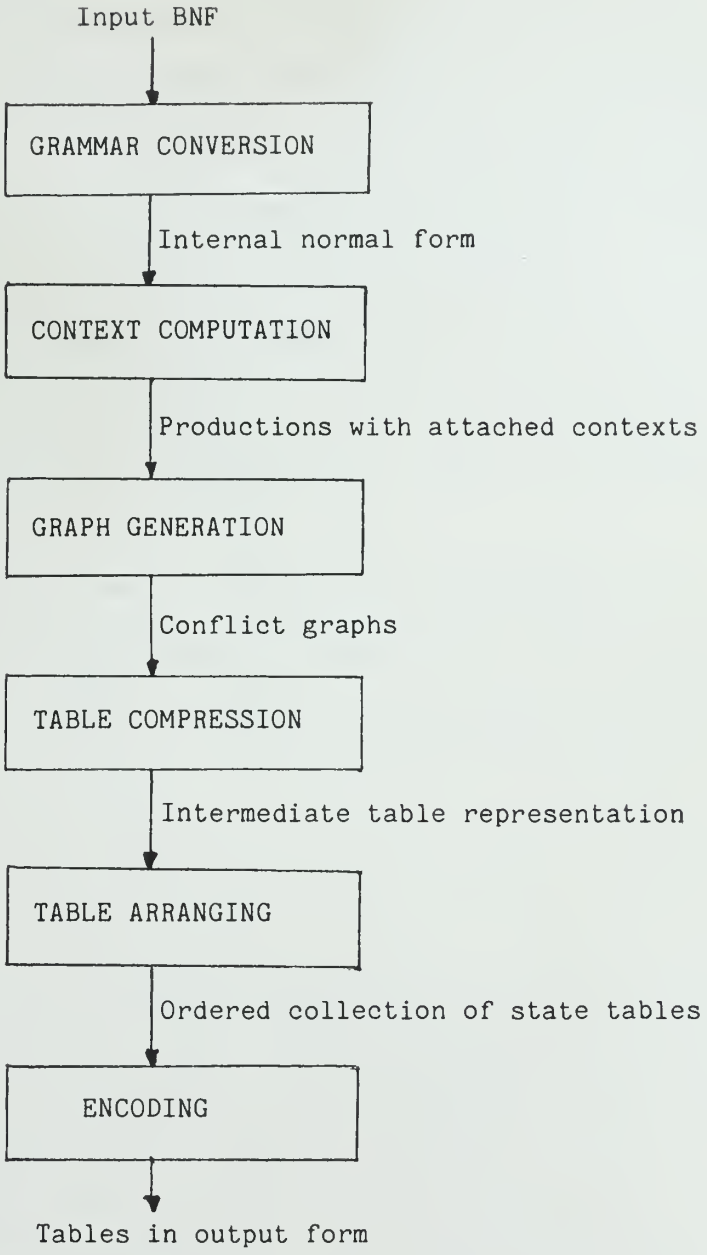
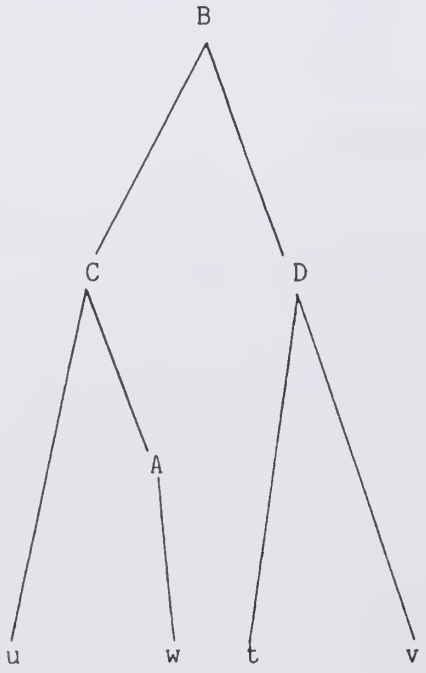

Input BNF
GRAMMAR CONVERSION
Internal normal form
CONTEXT COMPUTATION
Productions with attached contexts
GRAPH GENERATION
Conflict graphs
TABLE COMPRESSION
Intermediate table representation
TABLE ARRANGING
Ordered collection of state tables
ENCODING
Tables in output form

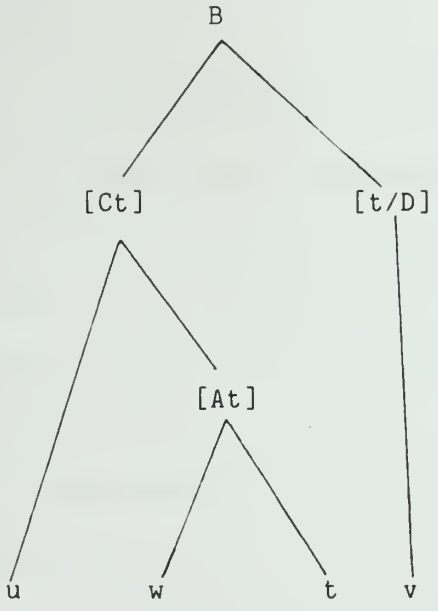
Figure 4.1

The transformation from the input grammar to internal normal form is accomplished in stages; each stage generates a grammar which generates the same language and which preserves all "sparse" parses. It is assumed that the initial grammar is LR(k) for some k. The LR(k) grammar is first transformed to an SLR(1) grammar; the SLR(1) grammar is then converted to a (1,1)BRC grammar; and finally this grammar is transformed to the internal normal form. If the input grammar is already known to be SLR(1) or (1,1)BRC, stages of the transformation can be bypassed.

The transformation from LR(k) to SLR(1) is based on the following concept. If A -> w is a rule which is <u>not</u> SLR(1), then the grammar is modified so that trees of the form



are transformed to the form




This process is known as look-ahead reduction. Look-ahead reduction consists of two transformations. The first is right-context extraction: if A is an offending rule and Bρ*A, then the grammar is said to be right-context extracted iff C -> uBy in P implies that either y is the empty string or y=av for some terminal, a; right context extraction is the conversion of a grammar to a right-context extracted form. The second transformation is premature scanning: if A -> w is an offending rule, $B_\rho$*A and C ->uBav is in P, then Ba is combined to [Ba], a non-terminal--the string Ba is replaced by [Ba] in all right parts in which it appears. Also, rules C -> wX, where Bρ*C are replaced by

[Ca] -> wXa if X is a terminal symbol and by [Ca] -> w[Xa] if X is a nonterminal. If a grammar is not SLR(1) then the look-ahead reduction transformations are applied to yield a new grammar. The process is repeated until the transformed grammar is finally SLR(1). If the original grammar is LR(k), then at most k applications of the transformation are required.

The SLR(1) grammar is subsequently transformed to (1,1)BRC using 'state splitting' and left stratification; for details, see [Graham 71]. The (1,1)BRC grammars are then transformed to the internal normal form by repeated left factorization. Given productions P = $\{A_i \rightarrow uB_iv\}$, left factorization replaces the set by P' = $\{A_i \rightarrow [u]B_iv\}$ and P' = {[u] -> u}; [u] is added to the nonterminals. Application is repeated as long as there is a factor of length bigger than 2. Eventually, all productions are of the proper form. However, there may be situations such that A -> BC and A ->DB (or A ->Ba). This is called a stack-state conflict; such conflicts can be removed by further application of left factorization, replacing A -> BC by A ->B'C and adding B' -> B.

## 4.4 CONTEXT COMPUTATION

### 4.4.1 ALGORITHMS

The second phase of the system involves the computation of the canonical mapping, W*. For the sake of convenience, the context sets are attached to the productions of the grammar. The productions for each state are kept in lists, so that at the end of the phase, the mapping is available for each state. The computation and collection of context sets is performed in three subphases: the first subphase generates the binary relations (and inverses) given in section 2.5; the second generates the first, follow and precede sets from these relations; and the last associates the context sets with the appropriate productions.

As we showed in 2.5, to compute the necessary context sets, it is sufficient to compute the first, follow, and precede sets for each nonterminal. This is done in two steps: first, $\lambda$ ,$\rho$ ,$\alpha$ and $\alpha^{-1}$ are computed; in the second, the transitive closures of and are computed and the relations composed to construct the sets. The first step is accomplished in one pass through the production rules. For each production, the partially computed sets, FIRST, FOLLOW, and PRECEDE and the binary relations are computed as follows:

1. For $p_i = A_i \rightarrow A_{i1}A_{i2}$, $A_i$ is added to $\lambda A_{i1}$ and to $\rho A_{i2}$ and $A_{i1}$ is added to PRECEDE($A_{i2}$).

2. For $p_i = A_i \rightarrow A_{i1}a_{i2}$, $A_i$ is added to $\lambda A_{i1}$ and $a_{i2}$ is added to FOLLOW($A_{i1}$).

3. For $p_i = A_i \rightarrow A_{i1}$, $A_i$ is added to $\lambda A_{i1}$ and to $\rho A_{i1}$.

4. For $p_i = A_i \rightarrow a_{i1}$, $a_{i1}$ is added to FIRST($A_i$).

In this step, we also construct a directed graph, the renaming graph representing the renaming rule structure of the grammar. The vertices of the graph are the nonterminals of the grammar. A labeled edge from $A_i$ to $A_{i1}$ is added for all renaming rules $p = A_i \rightarrow A_{i1}$. The label applied is the index of the production.

To complete the generation of the sets FIRST, FOLLOW, and PRECEDE, requires transitive closure. Closure is computed by the following $O(n^3)$ algorithm, where n is the number of nonterminals. (Actually, the algorithm is $O(n^2m)$ where m is the number of computer words required to store n bits.)

Input: A binary relation, $\beta$ , as an array of bit vectors.

Output: The transitive closure of $\beta$, $\beta^*$.

(1) Set $\beta^* x_i$ to $\{x_i\} \cup \beta x_i$ for all $x_i$.

(2) <u>for</u> i from 1 to n <u>do</u> for all $x_i \beta^* x_j$ include $\beta^* x_i$ in $\beta^* x_j$.

It is easy to verify that this algorithm is correct.

After the closures are computed, FIRST, FOLLOW, and PRECEDE are completed as follows:

1. FIRST(B) is included in FIRST(A) for all A $\lambda^+$ B.

2. PRECEDE(A) is included in PRECEDE(B) for all A$\lambda^+$ B.

3. FIRST(A) is included in FOLLOW(B) for all A, B such that C -> BA.

4. FOLLOW(A) is included in FOLLOW(B) for all A $\rho^+$B.

The context sets are attached to productions in a manner prescribed by the procedure given in 2.5. For each input erasing and stack renaming production, p, the appropriate context sets are determined and attached to the production; further, they are added to every production, p', which labels an edge in the renaming graph that is in any path from a nonterminal in the right part of p. Traversing the renaming graph is a simple process; there can be no cycles in the graph, nor can there be

multiple paths between vertices. Therefore, each nonterminal can be encountered at most once in any traversal of the renaming graph. Thus, distributing context sets is an O(mn) algorithm, where m is the number of input erasing and stack reducing rules and n the number of nonterminals.

### 4.4.2 REPRESENTING CONTEXT SETS

The context information for each production is recorded as an index into a table of context pairs. The pairs map into two tables of bit vectors, one table each for the left and right context sets. Each bit vector in a table is unique; when a new context set is generated during context computation, a hash addressing scheme is used to locate the corresponding bit vector; if it is not present, it is added to that table. The tables are initialized so that all the singleton subsets of N are included in the left context set table and all the singleton subsets of T are included in the right context set table. Entries with the same hash address are chained together. Figure 4.2 illustrates the layout of the context tables.

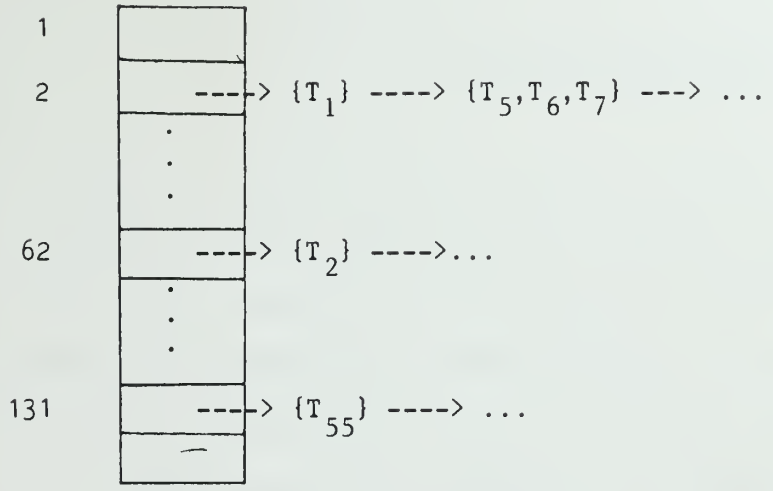



Figure 4.2

## 4.5 CONFLICT GRAPH GENERATION

The third phase generates conflict graphs for each state. The pair of graphs and the sets of left and right compression candidates is

generated for all productions in the list for the specific state. graphs are represented by the vertex set and the two adjacency relations; the adjacency relations are computed from the definition given in section 3.4. Each vertex is mapped into a triple consisting of a production rule index and indices into the context set tables.

## 4.6 STATE TABLE GENERATION

Phase four creates state tables from the graphs created in phase three. The tables are generated in three subphases. The first divides the table entries into two partitions: full entries and compressed entries. The second subphase orders the entries, and the third subphase performs the compression and merging transformations and generates an intermediate representation of the state table. In the implementation, the second and third subphases are carried out concurrently. For ease of presentation, these subphases will be treated as separate sequential steps.

The process used to partition the full and compressed entries is very simple. The following stucture is used.

Algorithm 4.5.1

Input: A set of vertices, X, and a pair of adjacency relations, $adj_L$ and $adj_R$.

Output: FULL, a set of full entries, and COMPRESS, a set of compressible entries.

(1) Let Y = X.

repeat

(2) Remove vertices from Y until there is a free vertex in the subset; add the vertices removed to FULL.

(3) Generate S, a set with no free vertices, by removing free vertices from Y until it is not possible to continue; add the vertices removed to COMPRESS.

(4) Let Y be the set of remaining vertices in S.

until S=ø.

The reduction from Y to S specified in step (2) can be achieved as follows. Alternately scan the remaining left and right compression candidates looking for isolated vertices on the induced subgraphs. The process terminates when none are left. The following algorithm is based on this principle.

Algorithm 4.5.2

Input: Y, a set of vertices, and the relations $adj_L$ and $adj_R$ on the elements of Y.

Output: A subset of Y containing no free vertices.

Method: Let $C_L$ and $C_R$ be the sets of left and right compression candidates; the sets L and R will be used to hold potential free vertices from $C_L$ and $C_R$. Vertices are removed from L and R during the scanning process and added to L and R when freeing a vertex could cause the vertex added to become isolated.

(1) Let $L = C_L$, $R = C_R$, and $S = Y$.

repeat

(2) /* left scan */ Pick a vertex, y, from L and remove it. If $adj_R y \cap S = \emptyset$, then remove y from S and add vertices from $adj_L y \cap C_R \cap S$ to R.

(3) /* right scan */ Pick a vertex, y, from R and remove it. If $adj_L y \cap S = \emptyset$, then remove y from S and add vertices from $adj_R y \cap C_L \cap S$ to L.

until $L \cup R = \emptyset$

This algorithm is O(e+n), where e is the total number of edges in both conflict graphs and n is the number of vertices. This is easy to see: a vertex is examined at least once and once additionally for every vertex adjacent to it removed in a left or right scan.

Given the output of this algorithm, it is necessary to find a good set of vertices to remove in order to obtain a free vertex. As we showed in 3.5, there are no known polynomial-time algorithms to generate an optimal compression set. Therefore, the algorithm presented here is only approximate; it functions by removing the cheapest immediate set which will produce a free vertex. In those cases in which there are few vertices to be removed, the algorithm is good if not exact.

Algorithm 4.5.3

Input: A set S containing no free vertex.

Output: A subset S' of S containing at least one free vertex.

(1) Let L = left compression candidates, R = right compression candidates.

(2) Find x in L s.t. $|adj_R x \cap S|$ is smallest.

(3) Find y in R s.t. $|adj_L y \cap S|$ is smallest.

(4) If $|adj_L y \cap S| < |adj_R x \cap S|$ then set $S = S - adj_L y$;
otherwise, set $S = S - adj_R x$.

When this algorithm is used as step 3 of algorithm 4.5.1, the result is the same as would be obtained from the backtracking algorithm 3.2 if it halted after finding the first compression set. Since algorithm 4.5.3 is $O(n)$, algorithm 4.5.1 is $O(n^2+ne)$.

Given that the partitions have been computed, it is necessary to order both sets. We will address only the problem of ordering the compressed entries here. We showed in section 3.5 that there is a polynomial algorithm to find an ordering which provides maximum exploitation of the unconditional entries; further, we gave a bound for the algorithm. In the current implementation a faster, albeit suboptimal, ordering algorithm is used. Time is saved by not repeating the ordering process for each renaming rule in the grammar; instead, the best choice is "predicted" and the algorithm defers outputting corresponding entries when it can. At the same time, the algorithm attempts to improve table size by setting up mergeable sequences of entries.

We present a slightly simplified version of the algorithm in use; additional detail does not add to the discussion. To simplify notation, let R(v) be the largest set containing vertex v and consisting of vertices that correspond to entries for the same rule.

Algorithm 4.5.4

Input: A compressible set, Y.

Output: An ordering of Y and a compression scheme s, a function from Y to the set {left, right, both, either}.

(1) Let L = left compression candidate set, and R = right compression candidate set.

repeat

(2) If there exists x in $(L-R) \cap Y$ s.t. $adj_R x \cap Y = \emptyset$, then output x, set s(x) = left, and remove x from Y.

(3) else if there is x in $(R-L) \cap Y$ s.t. $adj_L x \cap Y = \emptyset$ then output x, set s(x) = right, remove x from Y.

(4) else let $U_L = \{x \in L \cap R \mid adj_L x \cap Y = \emptyset\}$, let $U_R = \{x \in L \cap R \mid adj_R x \cap Y = \emptyset\}$, $U_1 = U_L \cup U_R - (U_L \cap U_R)$, and $U_2 = U_L \cap U_R$.

(5) If $U_1 \cup U_2 = \emptyset$, then output an error entry and halt.

(6) If $U_1 \neq \emptyset$, pick y in $U_1$ s.t. $|R(y) \cap Y|$ is smallest. If y is in $U_R$ ($U_L$) output $R(y) \cap U_R$ ($R(y) \cap U_L$) and for all x in $R(y) \cap U_R$ ($R(y) \cap U_L$) set s(x) = left (right) and remove x from $U_1$ and Y.

until $U_1 = \emptyset$.

repeat

(7) pick y in $U_2$ s.t. $|R(y) \cap Y|$ is smallest.

(8) if not $(R(y) \subseteq Y)$ then output $R(y) \cap Y$ and for all x in $R(y) \cap Y$, set s(x)=either and remove x from $U_2$ and Y.

until $R(y) \subseteq Y$.

(9) output Y and set s(y)=both.

The algorithm as given is clearly $O(n^2)$. By using the technique that was used in algorithm 4.5.1, the algorithm can be improved to O(n+e).

Since the algorithm given to generate a compression set is O(ne), it dominates all the computation.

The final step in generating the state tables is to transform the state tables using the compression and merging transformations. Compression is performed to take maximum advantage of merging: for cases in which there is a sequence of entries $x_{i1},x_{i2},\ldots,x_{in}$, all in $R(x_{i1})$, for which $s(x_{ij})$=both, the choice of compression is based on the result of merging the entries. If merging left compressed entries results in the generation of a 'new' set and merging the entries as right compressed entries would not, the latter choice is made, and vice-versa. Each entry in the set tables is tagged to indicate whether it has been referenced in a state table, and tagged entries are given high priority. In this way, profliferation of sets is avoided. Obviously, the choices made in one state affect the compression scheme used later. It is clearly possible to make optimal choices, but only at the expense of using a combinatoric algorithm over the set of all states.

## 4.7 ARRANGING

Before the state tables are encoded, several global "optimizations" can be performed. These reduce the size of the tables and allow the parser to run faster as well. We will categorize all of these optimizations under the heading of "arranging". There are three distinct operations treated under this heading: arranging the initial state table, arranging the state tables, and arranging the state and

input vocabularies.

The initial state table entries are arranged so that the table can be handled specially. Instead of performing a sequential table lookup, it is possible to access the entries directly. Recall that productions for the initial state are of the form $A_i \rightarrow a_{ij}$, where the $a_{ij}$ are all distinct. If the input symbols which are the right parts of the initial state rules are mapped one-to-one on an interval 1...n, it is trivial to arrange the initial state table so that the right context for the i-th entry is the i-th input symbol. This allows direct access; if a symbol is in the proper range, the action for the corresponding entry is performed. If the symbol is out of range, an error is detected. By using direct access, the time spent accessing entries in the initial state is reduced dramatically. Because the parser is frequently in the initial state, and because the remaining states generally have few entries, this transformation allows the parser to be time efficient as well as compact.

States are arranged to take advantage of the properties of unconditional entries. Consider an unconditional entry for which the state/stack symbol is a state and for which there is no semantic action. Since the only action performed by the parser for that given entry is a transition from one state to another, it is possible to eliminate the entry by concatenating the states. The second state is treated as an "entry point" of the first. To determine an ordering which allows the exploitation of this property, the following procedure is used. We

construct an inverse forest, the nodes of which represent states. For each state, B , in which the unconditional entry specifies a transition to a state B without a semantic action, define an edge from B to B . At each node there are clearly m>=0 incoming edges and n<=1 outgoing edges. For every node such that m>1, remove m-1 edges arbitrarily. The result of this process is a collection of disjoint chains; each chain specifies the ordering of a group of states. By arbitrarily ordering the chains and preserving the ordering within them, the state tables are arranged to take advantage of the special unconditional entries.

The final arranging scheme we consider is rearranging the stack and input vocabularies. This is done to reduce the space required to store the auxiliary tables of bit vectors. We have assumed that it is necessary to accomodate the largest possible set in establishing the length of the (fixed-length) bit vectors. However, if no set contains the last k elements of this largest set, it is possible to truncate the bit vectors and to specify the range of symbols in the sets. The symbols in the sets can be arranged so that any symbol not appearing in any set used is assigned a numeric encoding larger than a specified range. That range is given by $|\bigcup X_i|$, where the $X_i$ are left (or right) context sets. In arranging the input symbols, the encoding of the symbols used as right parts of initial state rules must be preserved. The remaining symbols can be rearranged.

## 4.8 ENCODING THE TABLES

The final phase of the PGS encodes the state tables, putting them into a form usable on the parser's host machine. The tables can be output as binary files, macro definitions, initialization code, or other convenient representation; without loss of generality, we will assume the output to be in the form of macro definitions.

Input to the encoding phase consists of three distinct groups of data. State tables are input as an array of sixtuples consisting of a context pair, a state/stack symbol, an acceptor action, a semantic action, and a compression mode. A separate table contains the locations of the state tables in the first array, and the lengths of the state tables. The third data group is the pair of context set tables.

Output consists of a sequence of pseudo-instructions, which can be thought of as machine instructions for the parsing machine. There are four primitive actions performed by the machine:

1. Verify the current input symbol and apply an action.

2. Verify the current top of stack symbol and apply an action.

3. Apply an action unconditionally.

4. Indicate an error has occurred and enter error mode.

For the first two types of psuedo, the action is applied only if the symbol is matched. If the symbol is not matched, the next psuedo-instruction is fetched and interpreted. The psuedo-instruction format is shown in figure 4.3.

| operation | verification index | reduction flag | transition flag | symbol/ address |
|---|---|---|---|---|

Figure 4.3

The operation field indicates one of the four instruction types already described. The verification index is an index into a table of bit vectors or the index of a symbol. (In order to avoid storing singleton sets, the index values 1...m, where m=|N| or |T|, are used to represent the sets {1}...{m} and the table entries 1...k are represented by index values m+1...m+k.) The reduction flag indicates whether the symbol verified should be erased. The transition flag specifies how the

symbol/address field should be used: as the address of a psuedo-instruction or as a symbol to add to the stack. For the third type of psuedo-instruction, the verification index and reduction flag are unused; for the error pseudo, only the operation and address fields are used, the latter to indicate the state in which the error occurred.

In the examples used in the appendix, the operation codes are given by INPUT, STACK, ANY, and ERROR. The reduce flag values are represented symbolically as RED (erase/reduce the symbol) and NORED (do not). The transition flag values are ADD (add to the stack) and GOTO (next instruction to perform is at the address specified).

To encode a full entry, we use three instructions. The first checks one context and conditionally "branches" to a substate in which the second context is checked. The third instruction affects a return to the main state to execute further instructions. For example, the state given at the end of section 3.3 is encoded in figure 4.4; semantic actions are not shown. Addresses are given in symbolic form.

In a three instruction sequence, if the stack is reduced or the input shifted, that action must be performed in the second instruction. Further, the semantic action is performed before any symbol is consumed.

```
S1:     INPUT,1,NORED,GOTO,S11
S12:    INPUT,1,RED,GOTO,S2
        INPUT,2,RED,GOTO,S3
        STACK,1,RED,GOTO,S2
        ERROR,1
S11:    STACK,2,RED,GOTO,S5
        ANY,GOTO,S12
```

Figure 4.4

# CHAPTER 5

# RESULTS AND CONCLUSIONS

## 5.1 RESULTS

The principal claim of this thesis has been that the PGS is a flexible system that produces compact parsers. The results presented here support that claim. Parsers produced by PGS are compared with those obtained using well-known, efficient systems. Parse table sizes are given for three grammars: a simple expression grammar (see Appendix A), and grammars for XPL and Pascal. Results for XPL are of particular interest, as they provide a basis for direct comparison with the LR parsers produced by Anderson, Eve and horning and by Joliat. Their methods are the most efficient reported in the literature. The technique of Anderson, et al produces a list structured parse table, while Joliat produces a sparse-matrix representation. Indirect comparison is also available; Anderson, et al give results for Algol W, a language comparable in size and complexity to Pascal. As an additional benchmark, results from a previous version of PGS are shown.

The first table provides a complete synopsis of the results obtained. The number of table entries as shown indicates the minimum number of entries produced using all transformations and assuming no semantic actions are required (thus allowing the elimination of some unconditional entries). The space required by the tables is given for three different situations. The first figure given is a measure of the space required when the smallest possible number of bits is used to encode the pseudo-instructions, and assumes semantic actions are not required. The second figure given is based on a fixed, 30-bit pseudo-instruction length. The third figure assumes that no unconditional entries can be eliminated. The size of the tables produced for a compiler is likely to fall between the second and third figures.

| | (**) | Bits | Bits | Bits | Stack sets | Input sets | (**) | Full entries |
|---|---|---|---|---|---|---|---|---|
| Pascal | 349 | 7712 | 11370 | 12300 | 13 | 3 | 31 | 1 |
| XPL | 163 | 3296 | 5010 | 5820 | 0 | 2 | 27 | 0 |
| AEG (*) | 13 | 130 | 390 | 450 | 0 | 0 | 2 | 0 |

Table 5.1

(*) AEG is a simple expression grammar

(**) Unconditional entries removed by concatenating states.

Table 5.2 shows the comparitive figures for the methods of Anderson, et al, Joliat, and the previous PGS version. Table sizes are shown in bytes. Additional space is required for interpreters; however, this space is roughly the same in all cases. The table sizes reported for the new PGS also include all space required for context sets.

| | PGS (old) | PGS (new) | Anderson | Joliat |
|---|---|---|---|---|
| XPL | 1023 | 627 | 1128 | 1966* |
| PASCAL | 2480 | 1421 | --- | --- |
| ALGOL W | --- | --- | 2434 | --- |
| AEG | 200 | 49 | --- | 232 |

Table 5.2

(*) Joliat indicates that by eliminating error entries and doing full matrix reduction, a table size of 1000 can be obtained.

The limited data indicates that the compression techniques used in the PGS can potentially produce parse tables 25-35% smaller than those produced by very efficient LR methods.

## 5.2 CONCLUSIONS

This thesis has described a powerful parser generating system that is capable of producing efficient parsers for a wide class of grammars, namely the class of LR grammars. The contribution made by the thesis research is the development of techniques to acheive very good table compression. The techniques given in chapter 4 are efficient in that they do not require an exorbitant amout of computing resources in exchange for producing parse tables. The resulting system is capable of generating parsers for a large group of languages and making them available on small machines, i.e. mini and microcomputers.

## 5.3 EXTENSIONS AND REFINEMENTS

There are several possible refinements to the system which have not yet been invesigated. Additional heuristics for the compression algorithms might be developed, although the payoff does not promise to be extensive. Other areas to be investigated are error recovery and parser speedup. The compression of the parse tables sacrifices some effectiveness recovering and/or repairing syntactic errors. A compromise technique might be developed which perform full context checks for critical production rules. Determinig such situations can be tied in with the table generation algorithms presented in this paper. To do this requires a thorough analysis of applicable error recovery strategies.

Another refinement involves improving the time efficiency of the parser by doing 'partial chain elimination', bypassing renaming rules which do not contribute to the semantic phase. For certain contexts, a long chain of renaming rules might be involved which have no associated semantic action (or perhaps only one). By detecting these contexts, it is possible to establish a direct transition and avoid superfluous parsing decisions.

The final area for which some investigation is suggested is the grammar transformation phase. Any optimization of the output grammar which can be made in this phase potentially carries over into the remaining phases. Similarly for error recovery and parser speedup.

# REFERENCES

[Aho 72] Aho, A.V. and Ullman, J.D. The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Englewodd Cliffs, NJ 1972.

[Aho 74] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass 1974.

[Anderson 73] Anderson, T, Eve, J, and Horning, J.J. Efficient LR(1) parsers. Acta Informatica 2(1973),12-39.

[DeRemer 71] DeRemer, F.L. Simple LR(k) grammars. Comm. ACM 14(1971),453-460.

[Eickel 63] Eickel, J., Paul, M., Bauer, F.L., and Samelson, K. A syntax controlled generator of formal language processors. Comm. ACM 6(1963),451-455.

[Evans 64] Evans, A. Jr. An ALGOL 60 compiler. Ann. Rev. Automatic Programming 4(1964),87-124.

[Feldman 68] Feldman, J.A. and Gries, D. Translator writing systems. Comm. ACM 11(1968),77-113.

[Floyd 61] Floyd, R.W. A descriptive language for symbol manipulation. J. ACM 8(1961),579-584.

[Floyd 63] Floyd, R.W. Syntactic analysis and operator precedence. J. ACM 10(1963),316-333.

[Floyd 64] Floyd, R.W. Bounded-context syntactic analysis.

Comm. ACM 7(1964),62-67.

[Graham 74] Graham, S.L. On bounded right context languages and grammars. SIAM J. Comput. 3(1974),224-254.

[Gray 72] Gray, J.N. and Harrison, M.A. On the covering and reduction problems for context-free grammars. J. ACM 19(1972),675-698.

[Harary 69] Harary, F. Graph Theory. Addison-Wesley, Reading, Mass 1969.

[Ichbiah 70] Ichbiah, J. and Morse, S. A technique for generating almost optimal Floyd-Evans productions of precedence grammars. Comm. ACM 13(1970),501-508.

[Joliat 73] Joliat, M.L. On the reduced matrix representation of LR(k) parser tables. University of Toronto, Computer Systems Research Group Tech. Rep. CSRG-28,1973.

[Karp 72] Karp, R.M. Reducibility among combinatorial problems, in: Complexity of Computer Computation, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York, NY (1972),85-104.

[Lalonde 71] Lalonde, W.R. An efficient LALR parser generator. University of Toronto Computer Systems Research Group Tech. Rep. CSRG-2,1971.

[McKeeman 71] McKeeman, W.M., Horning, J.J., and Wortman, D.B. A Compiler Generator. Prentice-Hall, Englewood Cliffs, NJ 1971.

[Mickunas 73] Mickunas, M.D. Techniques for compressing bounded right context acceptors, Doctoral Diss., Purdue U.,

West Lafayette, Ind. May 1973.

[Mickunas2 73] Mickunas, M.D. and Schneider, V.B. A parser-generating system for constructing compressed compilers. Comm. ACM 16(1973),669-676.

[Mickunas 76] Mickunas, M.D., Lancaster, R.L., and Schneider, V.B. Transforming LR(k) grammars to LR(1), SLR(1) and (1,1) bounded right context grammars. J. ACM 23(1976), 511-533.

[Pager 70] Pager, D. A solution to an open problem by Knuth. Information and Control 17(1970),462-473.

[Wirth 66] Wirth, N. and Weber, H. EULER: a generalization of Algol 60 and its formal description. Comm. ACM 9(1966)13-25,89-99.

# APPENDIX A

## AN ARITHMETIC EXPRESSION GRAMMAR

### 1 GRAMMARS

#### BNF

Note: nonterminal symbols in both grammars are given as (subscripted) upper case letters; all other symbols, with the exception of the metasymbol '->', are terminal symbols.

```
E -> E+T
E -> T
T -> T*F
T -> F
F -> i
F -> (E)
```

# THE NORMAL FORM GRAMMAR

$S \rightarrow J_1\$$

$J_1 \rightarrow J_2E$

$E \rightarrow X_1T$

$E \rightarrow T$

$T \rightarrow X_2F$

$T \rightarrow F$

$F \rightarrow P\ )$

$F \rightarrow i$

$P \rightarrow X_3E$

$X_1 \rightarrow E\ +$

$X_2 \rightarrow T\ *$

$X_3 \rightarrow (\ $

$J_1 \rightarrow \$$

2 CONTEXT COMPUTATION

Shown are the follow and precede sets which are needed; there are no class 1 sets (see 2.5), so first sets are not given.

FOLLOW

E { $,+,) }

T { $,*,+,) }

F same as above

P { ) }

$J_1$ { $ }

PRECEDE

E { $X_3$,$J_2$ }

T { $X_1$, $X_3$, $J_2$ }

F { $X_1$, $X_2$, $X_3$, $J_2$ }

P same as F

$X_1$ same as E

$X_2$ same as T

$X_3$ same as F

NORMAL FORM WITH ATTACHED CONTEXTS

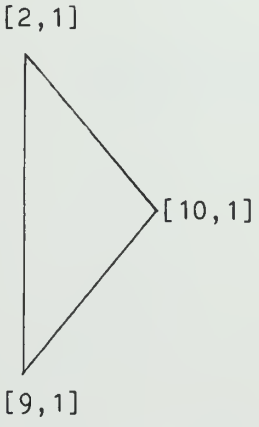
| | |
|---|---|
| $S \rightarrow J_1\$$ | |
| $J_1 \rightarrow J_2E$ | $( \{J_2\} X \{\$\} )$ |
| $E \rightarrow X_1T$ | $( \{X_1\} X \{\$,+,)\} )$ |
| $E \rightarrow T$ | $( \{J_2\} X \{\$\}, \{X_3\} X \{)\}, \{X_3,J_2\} X \{+\} )$ |
| $T \rightarrow X_2F$ | $( \{X_2\} X \{\$,*,+\} )$ |
| $T \rightarrow F$ | $( \{X_1\} X \{\$,+,)\}, \{J_2\} X \{\$\}, \{X_3\} X \{)\},$ $\{X_3,J_2\} X \{+\}, \{X_1,X_3,J_2\} X \{*\} )$ |
| $F \rightarrow P )$ | $( \{X_1,X_2,X_3,J_2\} X \{)\} )$ |
| $F \rightarrow i$ | $( \{X_1,X_2,X_3,J_2\} X \{i\} )$ |
| $P \rightarrow X_3E$ | $( \{X_3\} X \{)\} )$ |
| $X_1 \rightarrow E +$ | $( \{X_3,J_2\} X \{+\} )$ |
| $X_2 \rightarrow T *$ | $( \{X_1,X_3,J_2\} X \{*\} )$ |
| $X_3 \rightarrow ($ | $( \{X_1,X_2,X_3,J_2\} X \{(\} )$ |
| $J_1 \rightarrow \$$ | $( \{ \} X \{\$\} )$ |

3 CONFLICT GRAPHS

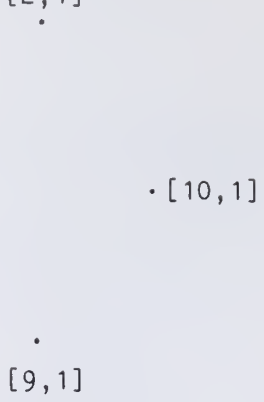
Conflict graphs are given for nontrivial cases; i.e. states E, T, and F. The vertices will be labeled by a pair consisting of the number of the production rule and the order of the context set in the collection of context sets for that production. For example, the entry $w^*(F,\{J_2\},\{\$\}) = \{(T,\text{rename})\}$ has a corresponding vertex numbered [6,2].

GRAPHS FOR STATE E

$G_{\underline{L}}$ =

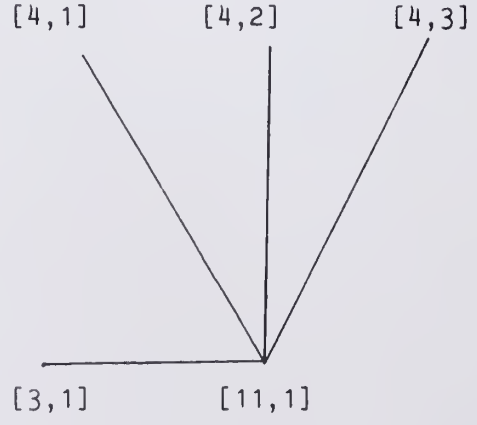


$G_R$ =

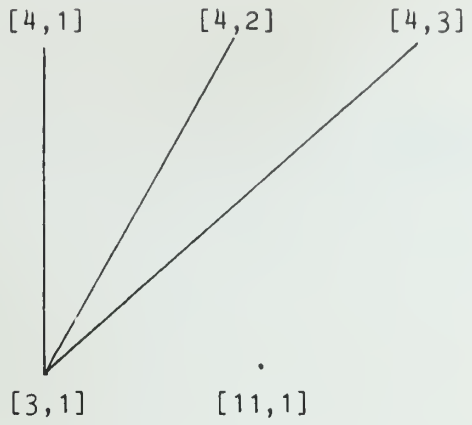



GRAPHS FOR STATE T

$G_L$ =

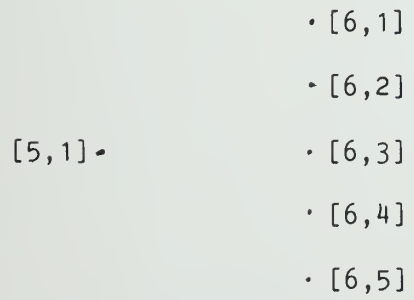


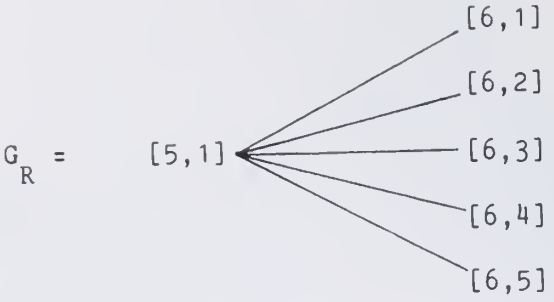
$G_R$ =




GRAPHS FOR STATE F

$G_L$ =




$G_R$ =
[5,1]
[6,1]
[6,2]
[6,3]
[6,4]
[6,5]

4 OUTPUT

The sequence shown is a symbolic representation of the parse tables produced for the simple arithmetic expression grammar. The minimum number of entries is shown.

```
S0: INPUT,$,RED,ADD,J
    INPUT,i,RED,GOTO,F
    INPUT,(,RED,ADD,X3
P : INPUT,),RED,GOTO,F
    ERROR,1
F : STACK,X2,RED,GOTO,T
T : INPUT,*,RED,ADD,X2
    STACK,X1,RED,GOTO,E
E : INPUT,+,RED,ADD,X1
    STACK,J2,RED,GOTO,J1
    ERROR,4
J1: INPUT,$,RED,GOTO,ACCEPT
    ERROR,5
```

# APPENDIX B

## PARSE TABLES FOR XPL

```
S0:     INPUT,ID,RED,GOTO,S13
        INPUT,CALL,RED,ADD,N15
        INPUT,IF,RED,ADD,N14
        INPUT,CHARACTER,RED,GOTO,S34
        INPUT,COMPLOP,RED,ADD,N16
        INPUT,NUMBER,RED,GOTO,S5
        INPUT,INITIAIL,RED,GOTO,S47
        INPUT,GOTO,RED,ADD,N1
        INPUT,STRING,RED,GOTO,S5
        INPUT,DO,RED,GOTO,S9
        INPUT,BIT,RED,GOTO,S48
        INPUT,LPAREN,RED,GOTO,S49
        INPUT,FIXED,RED,GOTO,S34
        INPUT,ADDOP,RED,ADD,N17
        INPUT,WHILE,RED,ADD,N24
        INPUT,DECLARE,RED,ADD,N18
        INPUT,LEFT,RED,ADD,N19
        INPUT,TO,RED,ADD,N20
        INPUT,SEMI,RED,GOTO,S4
        INPUT,LABEL,RED,GOTO,S34
```

```
        INPUT,RETURN,RED,GOTO,S26
        INPUT,CASE,RED,ADD,N21
S5:     STACK,N4,RED,GOTO,S52
S23:    STACK,N38,RED,GOTO,S31
S31:    INPUT,MULTOP,RED,ADD,N38
        STACK,N17,RED,GOTO,S2
        STACK,N31,RED,GOTO,S2
S2:     INPUT,ADDOP,RED,ADD,N31
        STACK,N37,NOR,GOTO,S45
S30:    INPUT,CONCAT,RED,ADD,N37
        STACK,N11,RED,GOTO,S20
        INPUT,RELOP,RED,ADD,N11
S20:    STACK,N16,RED,GOTO,S21
S21:    STACK,N36,RED,GOTO,S19
S19:    INPUT,ANDOP,RED,ADD,N36
        STACK,N35,NOR,GOTO,S46
S18:    INPUT,OROP,RED,ADD,N35
S12:    STACK,N5,RED,GOTO,S61
        STACK,N29,RED,GOTO,S59
        STACK,N39,RED,GOTO,S53
        STACK,N14,RED,GOTO,S50
        STACK,N12,RED,GOTO,S37
        STACK,N34,RED,GOTO,S16
        STACK,N24,RED,GOTO,S8
        STACK,N21,RED,GOTO,S8
        ERROR,12
```

```
S3:     STACK,N6,RED,GOTO,S3
S4:     STACK,N27,RED,GOTO,S4
        INPUT,ELSE,RED,GOTO,S32
S29:    STACK,N30,RED,GOTO,S28
        STACK,N3,RED,GOTO,S14
        STACK,N33,RED,GOTO,S14
S28:    INPUT,RIGHT,RED,GOTO,S25
        INPUT,END,RED,GOTO,S11
        ANY,GOTO,S30
S33:    INPUT,INITIAL,NOR,ADD,N13
S6:     STACK,N18,RED,GOTO,S7
        STACK,N32,RED,GOTO,S7
        ERROR,6
S7:     INPUT,COMMA,RED,ADD,N32
        INPUT,SEMI,RED,GOTO,S4
        ERROR,7
S8:     STACK,N22,RED,GOTO,S60
        ERROR,8
S9:     INPUT,SEMI,RED,ADD,N2
S22:    STACK,N28,RED,GOTO,S62
        ERROR,22
S11:    INPUT,ID,NOR,ADD,N25
S10:    STACK,N8,RED,GOTO,S58
        STACK,N2,RED,GOTO,S55
        ERROR,10
S13:    STACK,N7,RED,GOTO,S36
```

```
        STACK,N25,RED,GOTO,S10
        INPUT,LPAREN,RED,GOTO,S51
        INPUT,LITERALLY,RED,GOTO,S40
        INPUT,COLON,RED,GOTO,S17
        INPUT,INPUT01,NOR,ADD,N26
S35:    STACK,N1,RED,GOTO,S39
        STACK,N15,RED,GOTO,S38
        STACK,STACK01,NOR,GOTO,S27
        ANY,GOTO,S23
S14:    STACK,N27,RED,GOTO,S14
        ANY,GOTO,S29
S15:    STACK,N13,RED,GOTO,S33
        ERROR,15
S54:    INPUT,BY,NOR,GOTO,S56
S16:    STACK,N23,RED,GOTO,S8
        ERROR,16
S17:    INPUT,PROCEDURE,RED,GOTO,S24
        ANY,ADD,N27
S24:    INPUT,SEMI,RED,ADD,N8
        ANY,ADD,N28
S25:    STACK,N19,RED,GOTO,S1
        ERROR,25
S26:    INPUT,SEMI,RED,GOTO,S4
        ANY,ADD,N29
S27:    INPUT,EQUAL,RED,ADD,N34
        INPUT,COMMA,RED,ADD,N6
```

```
        ERROR,27
S32:    STACK,N3,RED,ADD,N33
        ERROR,32
S34:    STACK,N10,RED,GOTO,S33
        STACK,N26,RED,GOTO,S33
        ERROR,34
S36:    INPUT,RPAREN,RED,GOTO,S22
        INPUT,COMMA,RED,ADD,N7
        ERROR,36
S37:    INPUT,RPAREN,RED,GOTO,S35
        INPUT,COMMA,RED,ADD,N12
        ERROR,37
S38:    INPUT,SEMI,RED,GOTO,S4
        ERROR,38
S39:    INPUT,SEMI,RED,GOTO,S4
        ERROR,39
S40:    INPUT,STRING,GOTO,S6
        ERROR,40
S41:    INPUT,NUMBER,RED,GOTO,S42
        ERROR,41
S42:    INPUT,RPAREN,RED,GOTO,S34
        ERROR,42
S43:    INPUT,NUMBER,RED,GOTO,S63
        ERROR,43
S52:    INPUT,RPAREN,RED,GOTO,S15
S44:    INPUT,COMMA,RED,ADD,N4
```

```
        ERROR,44
S45:    STACK,N37,RED,GOTO,S30
        ERROR,45
S46:    STACK,N35,RED,GOTO,S18
        ERROR,47
S48:    INPUT,LPAREN,RED,GOTO,S41
        ERROR,48
S49:    STACK,N28,NOR,ADD,N7
        ANY,ADD,N5
S50:    INPUT,THEN,RED,ADD,N3
        ERROR,50
S51:    STACK,N52,NOR,GOTO,S43
        ANY,ADD,N12
S53:    INPUT,SEMI,NOR,GOTO,S57
        ANY,ADD,N23
S55:    INPUT,SEMI,RED,GOTO,S4
        ERROR,55
S56:    INPUT,BY,RED,ADD,N34
        ERROR,56
S57:    INPUT,SEMI,RED,GOTO,S3
        ERROR,57
S58:    INPUT,SEMI,RED,GOTO,S4
        ERROR,58
S59:    INPUT,SEMI,RED,GOTO,S4
        ERROR,59
S60:    INPUT,SEMI,RED,ADD,N2
```

```
        ERROR,60
S61:    INPUT,RPAREN,RED,GOTO,S23
        ERROR,61
S62:    INPUT,SEMI,RED,ADD,N8
        ERROR,62
S63:    INPUT,RPAREN,RED,ADD,N12
        ERROR,63
```

The sets used are:

```
STACK01 = { N2,N3,N6,N8,N19,N22,N27,N30,N33 }
STACK02 = { N18,N32 }
INPUT01 = { CHARACTER, BIT, FIXED, LABEL }
```

| BIBLIOGRAPHIC DATA SHEET | 1. Report No. UIUCDCS-R-77-904 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|

4. Title and Subtitle
A Method for Constructing Compressed Parsers For A Parser Generating System

5. Report Date
November, 1977

6.


7. Author(s)
Richard Marion Schell, Jr.

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Department of Computer Science
University of Illinois
Urbana, Illinois 61801


10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts


A parser generating system is described and technique is presented for constructing parse tables that are compact and that provide efficient parsing. The parse tables are constructed for a special class of grammars, which can be produced from LR(k) grammars using existing transformations. It is shown that producing optimal parse tables for the construction method is NP-complete; heuristic algorithms are given which are good in practice. Empirical data is presented to support this claim: tables are compared with those given for other parser generating systems and are shown to be smaller.


17. Key Words and Document Analysis. 17a. Descriptors


parsing, parser generating systems, table-driven parsers, bounded right context, context-free grammars, normal-form grammars, pushdown automata, grammatical transformations, NP-complete problems.


17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

| 18. Availability Statement | 19. Security Class (This Report) UNCLASSIFIED | 21. No. of Pages |
|---|---|---|
| | 20. Security Class (This Page UNCLASSIFIED | 22. Price |

FEB 8 1978

AUG 1 1979

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
3 0112 000447992